O'REILLY

TURING 图灵程序设计丛书



# Clojure 经典实例

Clojure Cookbook

[美] Luke VanderHart [加] Ryan Neufeld 著
王海鹏 徐宏宁 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

# 数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

TURING 图灵程序设计丛书

# Clojure经典实例

## Clojure Cookbook

### Recipes for Functional Programming

[美] Luke VanderHart [加] Ryan Neufeld 著
王海鹏 徐宏宁 译

O'REILLY®

*Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo*

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社

北 京

图书在版编目（CIP）数据

Clojure经典实例 / （美）范德哈特
（VanderHart,L.），（加）诺伊费尔德（Neufeld,R.）著；
王海鹏，徐宏宁译. -- 北京 : 人民邮电出版社，2015.8
（图灵程序设计丛书）
ISBN 978-7-115-39594-8

Ⅰ. ①C… Ⅱ. ①范… ②诺… ③王… ④徐… Ⅲ. ①
JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第131402号

## 内 容 提 要

本书以具体实例的形式讲解了Clojure语言在不同领域的应用，不仅介绍如何运用Clojure，而且还展示了很多常见库。书中给出了添加了注释的示例代码，详细分析并解释了数百个真实世界的编程任务。读者既可通过本书深入了解Clojure的精髓，也可将本书用作参考指南，解决具体问题。

本书适合各层次Clojure开发人员阅读。

◆ 著　　　[美] Luke VanderHart　[加] Ryan Neufeld
　译　　　王海鹏　徐宏宁
　责任编辑　岳新欣
　执行编辑　张　曼
　责任印制　杨林杰

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　印刷

◆ 开本：800×1000　1/16
　印张：26.5
　字数：626千字　　　　2015年 8 月第 1 版
　印数：1－3 500册　　　2015年 8 月北京第 1 次印刷

著作权合同登记号　图字：01-2014-6469号

定价：79.00元

**读者服务热线：(010)51095186转600　印装质量热线：(010)81055316**

**反盗版热线：(010)81055315**

**广告经营许可证：京崇工商广字第 0021 号**

# 版权声明

© 2014 by Cognitect, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media，Inc. and Posts & Telecom Press, 2015. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2014。

简体中文版由人民邮电出版社出版，2015。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

# O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

# 目录

# 译者序

## 编程语言习得

“熟悉与优雅正交。”<br>——Rich Hickey

约二十年前，我买过一本高等教育出版社出版的《LISP 语言》，作者是马希文、宋柔。可惜当年没有老师指导，自己水平不够，未能深入下去，只留下了一点模糊的印象：LISP 语言适用于人工智能，括号很多。

几年前，图灵公司的朋友送我一本《黑客与画家》，我连夜看完，重新燃起了对 LISP 的兴趣。我在书评中写道：“读完之后有一种想去学习 LISP 语言的冲动。一个不懂 LISP 的 Java 程序员，不是一个好的 C++ 程序员。”

现在，我终于找到了机会，开始学习 Clojure 这种运行在 JVM 上的 LISP 方言。经过一段时间的学习，我完全被它迷住了！

首先吸引我的是它的函数式编程特性。作为一个学习 C++ 和 Java 多年的程序员，我已习惯在程序中使用各种名词抽象，也就是领域术语，希望在程序中体现领域专家的思想和认识水平。而在 Clojure 编程中，虽然它也很适合领域抽象，但它的抽象程度更高，它希望达到数学家认识世界的水平。问题的开头通常是“给定一个无限序列……”，而常见的例子是如何实现斐波那契数列。

Leslie Lamport 说过，要将事情描述得清晰准确，人类发明的最好语言就是数学。这种对“表达的经济性”的追求，对于中国人是不陌生的。中国是诗歌的国度，而且古人对言简意赅的追求也有许多例子，比如“逸马杀犬于道”的故事。所以我觉得，LISP/Clojure 在精神上与有追求的中国程序员是契合的。

其次，它特别适合开发领域特定语言（DSL）。在 LISP 社区中流传着一个笑话，可以说明这一点：任何足够大的软件，最后都会实现一个半调子 LISP 解析器。LISP 的底层抽象极其简单，允许程序员设计更多的抽象，来描述这个世界。

学习一门新的语言，会改变学习者的思维方式。在面向对象编程时，我们更多关注单个对象。在函数式编程中，我们更多关注函数和集合。在工作中，不一定马上有机会使用 Clojure，但其中学到的思维方式，将对编程产生立竿见影的影响。在翻译本书时，我同时在用 Lua 开发项目，学习了 Clojure，让我能写出更简洁、更优雅的 Lua 代码。

学习新语言有这样一些原则：（1）专注于与你相关的内容；（2）从学习这门语言的第一天起，就把它当作你的交流方式；（3）当你听得懂别人在说什么时，就会不知不觉慢慢习得这门语言；（4）语言不是大量的知识积累，而更像一种生理训练；（5）心理状态和生理状态都很重要，要愉快和放松。对于模棱两可要有一定的容忍性，对于细枝末节不要过于纠结，因为那会把你逼疯。

本书提供了大量的例子，覆盖了日常编程领域的方方面面，正是学习 Clojure 的好读物。在翻译本书的过程中，我学到了很多，在此郑重推荐给大家。不足之处，还望大家指正。

王海鹏

2014 年秋

# 前言

本书的首要目标是提供中等长度的 Clojure 代码示例，超越基本知识，关注真实世界的日常应用程序（而不是概念或学术问题）。

与此前的许多其他 Clojure 书籍不同，本书的主题不是语言本身，或它的功能和能力。本书关注开发者面对的具体任务（不论他们使用哪种编程语言），展示如何用 Clojure 来解决这些具体问题。

因此，本书确实不是也做不到包罗万象，因为可能的问题示例有无限多个。但是，我们希望记录大多数程序员会经常遇到的一些比较常见的问题。通过归纳，读者将能够学到一些常见的模式、方法和技术，有助于他们为自己面对的问题设计解决方案。

## 本书如何写成

关于本书，你要了解一件重要的事情：它首先是团队协作的成果。它不是由一两个人写成的，甚至不是一个确定好的团队的成果。相反，它是 60 多个最优秀的 Clojure 程序员协作的结果，他们来自世界各地、各行各业。这些作者每天都在真实的场景中使用 Clojure：从航空航天到社交媒体，从银行业到机器人，从 AI 研究到电子商务。

因此，你会在提供的实例中看到许多差异。有些快速而简要，有些则内容更为丰富，针对 Clojure 的基本原理和实现提供了易于理解的深刻洞见。

我们希望兴趣各异的读者都能从本书中有所获。我们相信，它的用处不仅在于查找具体问题的解决方案，也在于考察 Clojure 能够提供的各种表达能力。在编辑提交的内容时，我们非常吃惊地发现很多概念和技术对我们来说也是新的，希望对读者来说也是新的。

我们在写作和编辑时还发现，要确定我们想介绍的内容的范围是一件很难的事情。每个实例都很棒，可以无限细分，进而涉及多个话题，而每个话题又值得写一个实例、一章甚至

一本书。但每个实例也需要保持独立。每个实例应该提供一些有用的、有价值的信息，让读者可以理解并消化。

我们真诚地希望自己很好地平衡了这些目标，也希望你觉得这本书有用而不乏味，内容深刻而不是艰深难懂。

# 读者对象

我们希望所有使用 Clojure 的人都能从本书中学到一些东西。有许多实例介绍的是真正基础的内容，初学者会觉得有用，但还有许多实例探讨的是专业话题，高级开发者会觉得有用，有助于他们开始实践。

但如果你是 Clojure 新手，这可能不是你要看的第一本书，至少不要只看这本书。本书介绍了许多有用的话题，但不像优秀的入门教材那样系统或完整。下面列出了一般的 Clojure 书籍，将它们作为前导教材或补充教材会很有帮助。

# 其他资源

本书内容并不全面，也永远不可能全面。有许多内容要讲，并且由于采用了面向任务的实例，自然就排除了有条理、叙述式地解释整个语言的特点和能力。

要更线性、彻底地了解 Clojure 及其特点，我们推荐下面的书。

- 《Clojure 编程：Java 世界的 Lisp 实践》（O’Reilly，2012），作者是 Chas Emerick、Brian Carper 和 Christophe Grand。这是一本全面的、用于一般目的的 Clojure 好书，关注语言和常见任务，面向 Clojure 的初学者。
- 《Clojure 程序设计（第 2 版）》（Pragmatic Bookshelf，2012），作者是 Stuart Halloway 和 Aaron Bedra。这是第一本关于 Clojure 的书，为 Clojure 语言提供了清晰全面的介绍和指导。
- *Practical Clojure*（Apress，2010），作者是 Luke VanderHart 和 Stuart Sierra。它简明扼要地解释了 Clojure 是什么，它的特点是什么。
- 《Clojure 编程乐趣》（Manning，2011），作者是 Michael Fogus 和 Chris Houser。这是一本比较高级的教材，真正深入到 Clojure 的主题和原理。
- *ClojureScript: Up and Running*（O’Reilly，2012），作者是 Stuart Sierra 和 Luke Vander Hart。虽然本书和这里列出的其他 Clojure 书籍主要或全部在探讨 Clojure 本身，但 ClojureScript（一种 Clojure 方言，能编译成 JavaScript）已经得到了相当的发展。这本书介绍了 ClojureScript，以及如何使用它，并探讨了 ClojureScript 和 Clojure 之间的相似与不同。

最后，你应该看看本书的源代码，它们可以从 GitHub 自由下载（https://github.com/clojure-cookbook/clojure-cookbook）。网上选择的实例比印刷版本更多，我们仍在接受新实例的“拉取请求”（pull request），也许某天会加入本书的下一版。

# 本书结构

本书的章节主要是依据主题对实例进行分组，而不是严格的分类。一个实例完全有可能适用于不止一个章节，在这种情况下，我们试着根据我们的猜测，将它放在大部分读者首先会去寻找的地方。

实例包含三个主要部分和一个次要部分：问题、解决方案、讨论和参阅。实例的问题陈述提出了任务或要克服的障碍。它的解决方案解决了问题，展示了特定的技术或库，能够高效地完成该任务。讨论完善了相关知识，探讨了解决方案和相关注意事项。最后，参阅部分向读者指出了一些附加的资源或相关实例，帮助你采用描述的解决方案。

## 各章简介

本书由以下几章构成。

- 第 1 章“原生数据”和第 2 章“复合数据”介绍了 Clojure 内建的原生和复合数据结构，解释了许多常见的（以及不太常见的）使用方式。
- 第 3 章“广义计算”包含了一些有用的主题，广泛适用于许多不同的应用领域和项目，从协议这样的 Clojure 特征，到可选的编程范式，例如用 `core.logic` 实现逻辑编程，或用 `core.async` 实现异步协作。
- 第 4 章“本地 I/O”包含了程序在运行时与本地计算交互的所有方式。这包括读写标准输入输出流，创建并操作文件，序列化和反序列化文件等。
- 第 5 章“网络 I/O 和 Web 服务”包含了类似第 4 章的主题，但探讨的是通过网络的远程通信。它包括的实例涉及各种网络通信协议和库。
- 第 6 章“数据库”展示了连接和使用各种数据库的技术和工具。特别关注了 Datomic 数据库，它共享了 Clojure 背后关于值、状态和标识的哲学，并扩展到了持久存储的领域。
- 第 7 章“Web 应用”深入探讨了 Clojure 最常见的应用领域：构建和维护动态网站。它全面介绍了 Ring(Clojure 中最流行的 HTTP 服务器库),以及 HTML 模板和渲染的工具。
- 第 8 章“性能与开发效率”解释了拥有 Clojure 程序之后还需要做些什么，介绍了打包、分发、性能剖析、日志的常见模式，将正在进行的任务与应用的生命周期关联起来。
- 第 9 章“分布式计算”关注云计算以及在重量级分布式数据处理中使用 Clojure。特别关注了 Cascalog，它是一个声明式 Clojure 接口，面向 Hadoop MapReduce 框架。
- 最后但同样重要的是第 10 章“测试”介绍了各种技术，来确保代码和数据的完整性和正确性：从传统的单元测试和集成测试，到更全面的产生式测试和模拟测试，甚至还有

可选的编译时验证，利用 `core.typed` 实现静态类型。

# 软件获取

若要按照本书的实例操作，你需要正确安装 Java 开发工具（JDK）和 Clojure 事实上的构建工具 Leiningen。我们推荐第 7 版 JDK，但至少需要第 6 版。对于 Leiningen，至少是第 2.2 版。

如果你还没安装 Java（或者希望升级），请访问 Java 下载页面（http://www.oracle.com/technetwork/java/javase/downloads/index.html），按提示下载并安装 Java JDK。

要安装 Leiningen，请遵循 Leiningen 网站（http://leiningen.org/）的安装指南。如果已经安装了 Leiningen，通过执行 `lein upgrade` 命令来取得最新版本。如果不熟悉 Leiningen，请访问使用指南（https://github.com/technomancy/leiningen/blob/stable/doc/TUTORIAL.md），了解更多信息。

你不需要手工安装的就是 Clojure 本身，因为 Leiningen 将随时根据需要，替你安装。要验证安装，就运行 `lein repl` 来检查 Clojure 的版本：

```
$ lein repl
# ...
user=> *clojure-version*
{:major 1, :minor 5, :incremental 1, :qualifier nil}
```

> 某些实例在 GitHub 上提供了一些在线材料。如果你的系统上没有安装 Git，请按照安装指南（https://help.github.com/articles/set-up-git）操作，以便能将 GitHub 代码库签出到本地。

某些实例（如数据库实例），需要进一步安装软件。在这种情况下，实例将包含安装工具的额外信息。

# 本书约定

在这本全是解决方案的书中，你会发现其中有不少代码。Clojure 源代码使用等宽字体，像这样：

```
(defn add
  [x y]
  (+ x y))
```

如果 Clojure 表达式被求值并返回，该值将以注释的形式给出，跟在一个箭头后面，就像它出现在命令行中一样：

```
(add 1 2)
```

```
;; -> 3
```

在合适的时候，代码示例可能略去或省略返回值注释。最常见的两种情况就是在定义函数/var时和缩短较长的输出时：

```
;; 这会返回 #'user/one，但你真的关心吗？
(def one 1)

(into [] (range 1 20))
;; -> [1 2 ... 20]
```

如果表达式产生输出到 STDOUT 或 STDERR，就会有注释说明（分别用 *out* 或 *error*），跟着就是每行输出的注释：

```
(do (println "Hello!")
    (println "Goodbye!"))
;; -> nil
;; *out*
;; Hello!
;; Goodbye!
```

## REPL会话

看到 REPL 驱动开发目前正在流行，因此本书就成为了 REPL 驱动的。REPL（读取、求值、打印、循环）是交互式的提示符，对表达式求值并打印出结果。Bash 提示符、irb 和 python 提示符都是 REPL 的例子。本书中几乎每个实例，都是为在 Clojure REPL 中运行而设计的。

虽然 Clojure REPL 传统上显示为 user=> ...，但本书希望读者能够复制粘贴实例中所有的例子，并看到标示的结果。因此，例子中省略了 user=> 并以注释的方式给出了输出，让事情变得更容易。如果你在计算机旁，这就特别有帮助：只要复制粘贴代码示例，不用担心遇到不能执行的代码。

如果例子只适用于 REPL 的环境，我们将保留传统的 REPL 风格（带上 user=>）。下面两个例子分别是只适用于 REPL 的例子和它的简化版本。

只适用于 REPL：

```
user=> (+ 1 2)
3
user=> (println "Hello!")
Hello!
nil
```

简化版本：

```
(+ 1 2)
```

```
;; -> 3

(println "Hello!")
;; *out*
;; Hello!
```

## 控制台/终端会话

控制台会话（例如，shell 命令）用等宽字体表示，行开始的美元符号（$）表示 shell 提示符。输出打印前面没有（$）：

```
$ lein version
Leiningen 2.0.0-preview10 on Java 1.6.0_29 Java HotSpot(TM) 64-Bit Server VM
```

命令行末的反斜杠（\）告诉控制台，命令将在下一行继续。

**我们的金童 `lein-try`**

Clojure 不以它的扩展标准库而闻名。不像 Perl 或 Ruby 这样的语言，Clojure 的标准库相对比较小。Clojure 选择了简单和强大。因此 Clojure 是一种有许多库的语言，但不是内建的库（好吧，Java 除外）。

因为本书中这么多解决方案都依赖于第三方库，所以我们开发了 `lein-try`（https://github.com/rkneufeld/lein-try）。Leiningen 是 Clojure 事实上的项目工具，`lein-try` 是 Leiningen（http://leiningen.org/）的一个小插件，让你快速而容易地尝试各种 Clojure 库。

要使用 `lein-try`，请确保安装了 Leiningen，然后将你的用户特性描述文件（~/.lein/profiles.clj）编辑成下面的样子：

```
{:user {:plugins [[lein-try "0.4.1"]]}}
```

现在，在项目内外都可以使用 **`lein try`** 命令来启动 REPL，访问任何你喜欢的库：

```
$ lein try clj-time
#...
user=>
```

长话短说：只要可能，在用第三方库的实例中，你会看到要求执行 `lein-try` 命令。在 3.4 节中，有一个用 `lein-try` 尝试实例的例子。

如果实例不能通过 `lein-try` 运行，我们会努力提供足够的指令，说明如何在你的机器上运行该实例。

# 排版约定

本书使用下面的字体约定。

- 楷体
  新术语第一次出现时使用楷体，目的是强调。

- 等宽字体
  用于函数名、方法名和参数，用于数据类型、类和命名空间，在例子中表示输入和输出，在正则文本中表示字面代码。

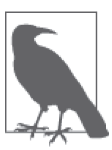
- 等宽黑体
  用于表示命令，你应该在命令行中照样输入。

- < 可取代的值 >
  路径、命令、函数名中的元素，应该由用户提供的值来取代尖括号及其中的内容。

库的名称符合两种惯例之一：具有适当名称的库用普通字体（如 Hiccup 或 Swing），而名称与代码符号相似的库用等宽字体（如 `core.async` 或 `clj-commons-exec`）。



这个符号表示提示或建议。



这个符号表示一般注释。



这个符号表示警告或注意。

# 使用代码示例

补充材料（代码示例、练习等）可以在 https://github.com/clojure-cookbook/clojure-cookbook 下载。

本书的目的是帮助你完成工作。一般来说，如果示例代码出现在本书中，就可以在你的程序和文档中使用它，不需要联系我们获得许可，除非你打算复制大量的代码。例如，写一个程序，用到本书中的几段代码，不需要获得许可。而销售或分发 O’Reilly 图书的示例

CD-ROM，确实需要许可。回答问题时摘录本书并引用示例代码，不需要获得许可。在你的产品文档中包含本书的大量示例代码，则确实需要许可。

我们感谢你说明来源，但这不是必须的。来源说明通常包括标题、作者、出版商和 ISBN。例如：“Luke VanderHart 和 Ryan Neufeld 所著 *Clojure Cookbook* (O’Reilly). Copyright 2014 Cognitect, Inc., 978-1-449-36617-9.”

如果你觉得你对代码的使用超出了合理的范围或上述允许的情况，可随时联系我们：permissions@oreilly.com.

# Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O’Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

# 联系我们

请把对本书的意见和疑问发送给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O’Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/clojure-ckbk

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

# 致谢

如果没有 Clojure 社区中许多人的无私奉献，本书不可能写成。超过 65 个 Clojure 开发者响应号召，提交实例，审读，并为本书的方向提供了建议。归根到底，这是一本属于社区的书，我们只是很荣幸能够将内容组织到一起。这些贡献者是：

- Adam Bard，adambard on GitHub
- Alan Busby，thebusby on GitHub
- Alex Miller，puredanger on GitHub
- Alex Petrov，ifesdjeen on GitHub
- Alex Robbins，alexrobbins on GitHub
- Alex Vzorov，0rca on GitHub
- Ambrose Bonnaire-Sergeant，frenchy64 on GitHub
- arosequist
- Chris Allen，bitemyapp on GitHub
- Chris Ford，ctford on GitHub
- Chris Frisz，cjfrisz on GitHub
- Clinton Begin，cbegin on GitHub
- Clinton Dreisbach，cndreisbach on GitHub
- Colin Jones，trptcolin on GitHub
- Craig McDaniel，cpmcdaniel on GitHub
- Daemian Mack，daemianmack on GitHub
- Dan Allen，mojavelinux on GitHub
- Daniel Gregoire，semperos on GitHub
- Dmitri Sotnikov，yogthos on GitHub

- Edmund Jackson，ejackson on GitHub
- Eric Normand，ericnormand on GitHub
- Federico Ramirez，gosukiwi on GitHub
- Filippo Diotalevi，fdiotalevi on GitHub
- fredericksgary
- Gabriel Horner，cldwalker on GitHub
- Gerrit，gerritjvv on GitHub
- Guewen Baconnier，guewen on GitHub
- Hoàng Minh Thắng，myguidingstar on GitHub
- Jason Webb，bigjason on GitHub
- Jason Wolfe，w01fe on GitHub
- Jean Niklas L'orange，hyPiRion on GitHub
- Joey Yang，joeyyang on GitHub
- John Cromartie，jcromartie on GitHub
- John Jacobsen，eigenhombre on GitHub
- John Touron，jwtouron on GitHub
- Joseph Wilk，josephwilk on GitHub
- jungziege
- jwhitlark
- Kevin Burnett，burnettk on GitHub
- Kevin Lynagh，lynaghk on GitHub
- Lake Denman，ldenman on GitHub
- Leonardo Borges，leonardoborges on GitHub
- Mark Whelan，mrwhelan on GitHub
- Martin Janiczek，Janiczek on GitHub
- Matthew Maravillas，maravillas on GitHub
- Michael Fogus，fogus on GitHub
- Michael Klishin，michaelklishin on GitHub
- Michael Mullis，mmullis on GitHub
- Michael O'Church，michaelochurch on GitHub
- Mosciatti S.，siscia on GitHub
- nbessi
- Neil Laurance，toolkit on GitHub
- Nurullah Akkaya，nakkaya on GitHub
- Osbert Feng，osbert on GitHub
- Prathamesh Sonpatki，prathamesh-sonpatki on GitHub
- R. T. Lechow，rtlechow on GitHub

- Ravindra R. Jaju，jaju on GitHub
- Robert Stuttaford，robert-stuttaford on GitHub
- Russ Olsen，russolsen on GitHub
- Ryan Senior，senior on GitHub
- Sam Umbach，sumbach on GitHub
- Sandeep Nangia，nangia on GitHub
- Steve Miner，miner on GitHub
- Steven Proctor，stevenproctor on GitHub
- temacube
- Tobias Bayer，codebrickie on GitHub
- Tom White，dribnet on GitHub
- Travis Vachon，travis on GitHub
- Stefan Karlsson，zclj on GitHub

最大的贡献者值得特别感谢：Adam Bard、Alan Busby、Alex Robbins、Ambrose Bonnaire-Sergeant、Dmitri Sotnikov、John Cromartie、John Jacobsen、Robert Stuttaford、Stefan Karlsson 和 Tom Hicks。这些杰出的开发者一起几乎贡献了本书三分之一的实例。

感谢我们的技术复查者 Alex Robbins、Travis Vachon 和 Thomas Hicks。在大约 11 个小时或更短的时间内，这几位先生查遍了本书，寻找技术错误。普通的技术复查者只会提交文本的问题描述，这几位则做得更多，常常提交拉取请求（pull request），修复了他们报告的所有错误。总之，和他们一起工作很开心，因为他们的参与，本书变得好了很多。

最后，感谢我们的雇主 Cognitect，让我们有时间完成本书，同时感谢所有的同事，他们提出了建议和反馈，而且最棒的是，他们提供了更多的实例！

## Ryan Neufeld

首先，非常感谢 Luke，是他最先提出了这本书的主意。我非常感激他邀请我加入，一起编写。人们说学习某样东西最好的方法就是写一本关于它的书，此言不虚。编写这本书确实丰富了我的 Clojure 技能，使我的水平提升到了另一个层次。

而且最重要的是，我要感谢家人，他们容忍我完成写书的过程。让这件事顺利起步是无比艰巨的任务，没有妻子 Jackie 和女儿 Elody 的爱和支持，这不可能完成。如果不是侵占了她们无数的夜晚、周末和休假时间，我不可能编写完这本书。

## Luke VanderHart

首先，我要感谢合著者 Ryan，他工作非常努力，参与编写了这本书。

同时，我在 Cognitect 的同事提供了许多想法和思路，最重要的是有一个很好的委员会，探讨在编写和编辑过程中出现的许多问题。非常感谢他们，同时也感谢他们提供机会，让我整天写 Clojure 代码，天天如此。

第 1 章

# 原生数据

## 1.0 简介

对于处理困难的问题，Clojure 是一门极好的语言。它的简单工具让软件开发者一层一层地建立抽象，直到能够轻松地处理世界上最难的一些问题。像化学一样，每个了不起的 Clojure 程序都归结为简单的原子，即原生类型。

很久以前，Clojure 就站在 Java 巨人的肩上，它利用了 Java 虚拟机（JVM）[1] 中提供的一组极好的类型，这些类型经过了实践的检验：字符串、数值类型、日期、通用唯一标识符（UUID），只要说得出的，Clojure 都有。本章探讨 Clojure 的原生类型，以及如何完成常见任务。

### 字符串

几乎所有编程语言都知道如何处理字符串，Clojure 也不例外。除了一些差别之外，Clojure 提供了像大多数其他语言一样的能力。下面是一些应该了解的关键差别。

首先，Clojure 字符串基于 Java 的 UTF-16 字符串。不需要在文件中添加注释来说明字符串的编码方式，也不需要担心在转换过程中丢失了字符。Clojure 程序已经准备好与英文字符之外的世界通信。

注 1：JVM 是执行 Java 字节码的地方。Clojure 编译器以 JVM 为目标，生成能运行的字节码。因此，你可以任意使用所有原生 Java 类型。

其次，Clojure 不像 Perl 或 Ruby 拥有较大的字符串程序库，其内建的字符串操作库相当精练。初看起来这可能有点奇怪，但 Clojure 喜欢简单的、可组合的工具，Clojure 中有许多集合操作函数，都能很好地处理字符串，因为它们也是集合！由于这个原因，Clojure 的字符串库小得出人意料。在 `clojure.string` 命名空间中，可以找到很小一组专门针对字符串的函数。

Clojure 也利用了它的宿主平台（JVM），没有重复 `java.lang.String` 类已实现的功能。在 Clojure 中使用 Java 互操作并不是一种失败的尝试，因为语言的设计就是为了便于互操作，使用内建的字符串方法通常和调用 Clojure 的函数一样方便。

我们建议在必要的时候，“require as” `clojure.string` 命名空间。盲目地 `:use` 一个命名空间总是令人气恼的 [2]，常常导致冲突或混乱。所以我们更喜欢给它取别名为 `str` 或 `s`，而非在所有东西前面加上 `clojure.string`，那有点奇怪：

```
(require '[clojure.string :as str])

(str/blank? "")
;; -> true
```

## 数值类型

对于数值类型，Clojure 和 Java 之间的差异比较大。但这不一定是坏事。虽然 Java 的数值类型可能非常快或具有任意的精度，但数值整体上没有一组精美的接口。Clojure 把 Java 的各种数值类型统一成为一致的包，每个困难的地方都有解决的办法。

本章中关于数值类型的实例，将展示如何利用这些设计，实现期望的速度、精度或表达能力。

## 日期

在 Java 生态系统中，日期和时间的历史长而曲折。需要 `Date`、`Time`、`DateTime` 或 `Calendar` 吗？谁知道呢。为什么这些 API 都那么不稳定？本章中的实例应该能够阐明何时使用恰当的内建类型，如何使用，以及若内建类型不够用（或者非常难用），何时去寻找外部库。

# 1.1 改变字符串的大小写

作者：Ryan Neufeld

注 2：用了 `use`，就在项目的命名空间中引入了许多新的符号，又没有留下线索表明它们来自哪里。这通常让代码维护者感到困惑和沮丧。我们强烈建议不要用 `use`。

## 问题

需要改变一个字符串的大小写。

## 解决方案

用 `clojure.string/capitalize` 来大写字符串中的第一个字符。

```
(clojure.string/capitalize "this is a proper sentence.")
;; -> "This is a proper sentence."
```

如果需要改变所有字符的大小写，请用 `clojure.string/lower-case` 或 `clojure.string/upper-case`：

```
(clojure.string/upper-case "loud noises!")
;; -> "LOUD NOISES!"

(clojure.string/lower-case "COLUMN_HEADER_ONE")
;; -> "column_header_one"
```

## 讨论

大小写函数只影响字母。虽然函数 `capitalize`、`lower-case` 和 `upper-case` 可能会改动字母，但标点符号或数字会保持不变：

```
(clojure.string/lower-case "!&$#@#%^[]")
;; -> "!&$#@#%^[]"
```

Clojure 对所有字符串都使用 UTF-16 编码，因此它对什么是字母的定义是相当宽泛的，包括有重音的字母。例如短句“Hurry up, computer!”，它包含字母 e，翻译成法语时会有锐音（é）和长音（ê）记号。由于这些特殊的字符都被视为字母，大小写函数可以对它们进行相应的改变：

```
(clojure.string/upper-case "Dépêchez-vous, l'ordinateur!")
;; -> "DÉPÊCHEZ-VOUS, L'ORDINATEUR!"
```

## 参阅

- `clojure.string` 命名空间的 API 文档（http://clojure.github.io/clojure/clojure.string-api.html）。
- `java.lang.String` 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/lang/String.html）。

# 1.2 清除字符串中的空白字符

作者：Ryan Neufeld

## 问题

需要清除字符串中的空白字符。

## 解决方案

使用 `clojure.string/trim` 函数来删除字符串首尾的所有空白字符：

```
(clojure.string/trim " \tBacon ipsum dolor sit.\n")
;; -> "Bacon ipsum dolor sit."
```

要处理字符串内部的空白字符，需要有点创造性。使用 `clojure.string/replace` 来修正字符串内部的空白字符：

```
;; 将空白字符压缩为一个空格
(clojure.string/replace "Who\t\nput all this\fwhitespace here?" #"\s+" " ")
;; -> "Who put all this whitespace here?"

;; 将 Windows 风格的换行替换成 Unix 风格的换行
(clojure.string/replace "Line 1\r\nLine 2" "\r\n" "\n")
;; -> "Line 1\nLine 2"
```

## 讨论

什么构成了 Clojure 中的空白字符？回答取决于功能：有些比另一些更自由，但可以放心地假定空格（ ）、制表符（`\t`）、换行（`\n`）、回车（`\r`）、走行（`\f`）和垂直制表符（`\x0B`）都会被当成空白字符。在 Java 的正则表达式实现中，这一组字符由 `\s` 匹配。

Ruby 和其他语言将字符串操作函数放在核心命名空间，Clojure 不同，它将 `clojure.string` 命名空间放在 `clojure.core` 之外，因此不能够直接使用。常用的技巧是将 `clojure.string` 引入为 `str` 或 `string` 这样的简写形式，让代码更简明：

```
(require '[clojure.string :as str])
(str/replace "Look Ma, no hands" "hands" "long namespace prefixes")
;; -> "Look Ma, no long namespace prefixes"
```

有时候，也许不需要把字符串两边的空白字符都删掉。如果只是想删除字符串左边或右边的空白字符，请分别使用 `clojure.string/triml` 或 `clojure.string/trimr`。

```
(clojure.string/triml " Column Header\t")
;; -> "Column Header\t"

(clojure.string/trimr "\t\t* Second-level bullet.\n") ,
;; -> "\t\t* Second-level bullet."
```

## 参阅

- 1.3 节“利用部件构建字符串”。

# 1.3 利用部件构建字符串

作者：Ryan Neufeld

## 问题

有多个字符串、值或集合，需要合并成一个字符串。

## 解决方案

使用 `str` 函数来连接多个字符串和（或）值：

```
(str "John" " " "Doe")
;; -> "John Doe"

;; str 也适用于变量，或其他任何值
(def first-name "John")
(def last-name "Doe")
(def age 42)

(str last-name ", " first-name " - age: " age)
;; -> "Doe, John - age: 42"
```

使用 `apply` 带 `str`，将值的集合连接成一个字符串：

```
;; 将一系列字符还原成一个字符串
(apply str "ROT13: " [\W \h \y \v \h \f \ \P \n \r \f \n \e])
;; -> "ROT13: Whyvhf Pnrfne"

;; 或者将一些行还原成一个文件（如果行都有换行符）
(def lines ["#! /bin/bash\n", "du -a ./ | sort -n -r\n"])
(apply str lines)
;; -> "#! /bin/bash\ndu -a ./ | sort -n -r\n"
```

## 讨论

Clojure 的 `str` 就像一个好的 Unix 工具：它做一件事，做得很好。如果向 `str` 提供一个或多个参数，它会对参数调用 Java 的 `.toString()` 方法，将每个结果加在后面。如果提供 `nil` 作为参数或不带参数调用，`str` 会返回代表字符串身份的值，即空串。

对于字符串连接，Clojure 采用了相当自由的方式。（`apply str ...`）没有什么专门针对字符串的东西。它只是使用了高阶函数 `apply`，模拟用变长参数调用 `str`。

这个 `apply`：

```
(apply str ["a" "b" "c"])
```

在功能上等价于：

```
(str "a" "b" "c")
```

既然 Clojure 在连接字符串时没有什么限制，我们就可以自由发挥，利用 Clojure 提供的大量操作函数。例如，从一行抬头和几行数据中构造逗号分隔的值（CSV）。这个例子特别适合 apply，因为可以在前面加上抬头，不用将它插在 rows 集合的前面：

```
;; 利用一行抬头字符串和几行数据来构造 CSV
(def header "first_name,last_name,employee_number\n")
(def rows ["luke,vanderhart,1","ryan,neufeld,2"])

(apply str header (interpose "\n" rows))
;; -> "first_name,last_name,employee_number\nluke,vanderhart,1\nryan,neufeld,2"
```

如果要做的事情不是太特别，apply 和 interpose 可能有些繁琐。要连接简单的字符串，通常用 clojure.string/join 更容易。join 函数接受一个集合和一个可选的分隔符。带分隔符时，join 返回的字符串是集合的所有元素，中间用该分隔符分隔。不带分隔符时，它返回所有元素挤在一起的字符串，类似于（apply str coll）的返回：

```
(def food-items ["milk" "butter" "flour" "eggs"])
(clojure.string/join ", " food-items)
;; -> "milk, butter, flour, eggs"

(clojure.string/join [1 2 3 4])
;; -> "1234"
```

## 参阅

- 1.6 节“格式化字符串”。
- clojure.string 命名空间的 API 文档（http://clojure.github.io/clojure/clojure.string-api.html）。
- java.lang.String 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/lang/String.html）。

# 1.4　将字符串作为字符序列

作者：Ryan Neufeld

## 问题

需要处理字符串中的单个字符。

## 解决方案

对字符串使用 seq，得到它包含的字符序列：

```
(seq "Hello, world!")
;; -> (\H \e \l \l \o \, \space \w \o \r \l \d \!)
```

但是，并非每次想处理字符串中的字符时，都要调用 seq。以序列为参数的所有函数，都会自动将字符串强制转换成字符序列：

```
;; 计算每个字符在字符串中出现的次数
(frequencies (clojure.string/lower-case "An adult all about A's"))
;; -> {\space 4, \a 5, \b 1, \d 1, \' 1, \l 3, \n 1, \o 1, \s 1, \t 2, \u 2}

;; 字符串中的每个字母都是大写的吗？
(defn yelling? [s]
  (every? #(or (not (Character/isLetter %))
               (Character/isUpperCase %))
        s))

(yelling? "LOUD NOISES!")
;; -> true

(yelling? "Take a DEEP breath.")
;; -> false
```

## 讨论

在计算机科学中，“字符串”意味着“字符序列”，Clojure 对字符串就是这么处理的。因为 Clojure 字符串背后就是字符序列，所以在需要集合的地方，都可以用字符串替代。如果这样做，字符串就会被解释为一个字符集合。(seq string) 没有什么特别的。seq 函数只是返回一个字符集合的序列，这些字符组成了这个字符串。

更常见的是，在对字符串中的字符做了某种工作之后，希望将这个集合恢复成一个字符串。对字符集合使用 apply 和 str，将它们还原成字符串：

```
(apply str [\H \e \l \l \o \, \space \w \o \r \l \d \!])
;; -> "Hello, world!"
```

## 参阅

- 1.3 节“利用部件构建字符串”。
- 1.5 节“字符与整数的转换”。

# 1.5 字符与整数的转换

作者：Ryan Neufeld

## 问题

需要将字符转换成对应的 Unicode 编码值（整数值），或反过来。

## 解决方案

用 `int` 函数将字符转换成它的整数值：

```
(int \a)
;; -> 97

(int \ø)
;; -> 248

(int \α) ; 希腊字母 alpha
;; -> 945

(int \u03B1) ; 希腊字母 alpha（按编码值）
;; -> 945

(map int "Hello, world!")
;; -> (72 101 108 108 111 44 32 119 111 114 108 100 33)
```

用 `char` 函数返回整数编码值对应的字符：

```
(char 97)
;; -> \a

(char 125)
;; -> \}

(char 945)
;; -> \α

(reduce #(str %1 (char %2))
        ""
        [115 101 99 114 101 116 32 109 101 115 115 97 103 101 115])
;; -> "secret messages"
```

## 讨论

Clojure 继承了 JVM 强大的 Unicode 支持。所有字符串都是 UTF-16 字符串，所有字符都是 Unicode 字符。前面 256 个 Unicode 编码值与 ASCII 码相等，这很方便，让标准的 ASCII 文本很容易处理。但是，Clojure 像 Java 一样，没有对 ASCII 码进行任何特殊处理，字符和整数之间的一一对应表明，编码值直接延伸到整个 Unicode 空间。

例如，表达式 `(map char (range 0x0410 0x042F))` 列出所有的斯拉夫语大写字母，它们处于 Unicode 的这个范围：

```
(\А \Б \В \Г \Д \Е \Ж \З \И \Й \К \Л \М \Н \О \П \Р \С \Т \У \Ф
\Х \Ц \Ч \Ш \Щ \Ъ \Ы \Ь \Э \Ю)
```

`char` 和 `int` 函数的主要用处，是将一个数字强制转换成 `java.lang.Integer` 或 `java.lang.Character` 的实例。`Integer` 和 `Character` 最终都是数字编码的，尽管 `Character` 还支持一

些额外的文本相关方法，要先转换成真正的数字类型，才能用于数学表达式。

## 参阅

- *Unicode Explained*（http://oreil.ly/unicode-explained），作者 Jukka K. Korpela (O’Reilly)，真正全面地探讨了 Unicode 和国际化的工作原理。
- 1.4 节“将字符串作为字符序列”，详细讨论了处理构成字符串的字符。
- 1.15 节“解析数字”。

# 1.6 格式化字符串

作者：Ryan Neufeld

## 问题

需要在字符串中插入一些值，并设定这些值在字符串中出现的格式。

## 解决方案

将值格式化后插入字符串的最快方法，是 `str` 函数：

```
(def me {:first-name "Ryan", :favorite-language "Clojure"})
(str "My name is " (:first-name me)
     ", and I really like to program in " (:favorite-language me))
;; -> "My name is Ryan, and I really like to program in Clojure"

(apply str (interpose " " [1 2.000 (/ 3 1) (/ 4 9)]))
;; -> "1 2.0 3 4/9"
```

但使用 `str` 时，值的插入是不加思考的，以默认的 `.toString()` 方式显示。不仅如此，有时候看着 `str` 的格式，很难解释想要的输出是什么。

要更好地控制这些值的显示方式，请用 `format` 函数：

```
;; 产生一个文件名，带有 0 补全的可排序的索引
(defn filename [name i]
  (format "%03d-%s" i name)) ; ❶

(filename "my-awesome-file.txt" 42)
;; -> "042-my-awesome-file.txt"

;; 创建一个对齐的表格
(defn tableify [row]
  (apply format "%-20s | %-20s | %-20s" row)) ; ❷

(def header ["First Name", "Last Name", "Employee ID"])
```

```
(def employees [["Ryan", "Neufeld", 2]
                ["Luke", "Vanderhart", 1]])

(->> (concat [header] employees)
     (map tableify)
     (mapv println))
;; *out*
;; First Name           | Last Name            | Employee ID
;; Ryan                 | Neufeld              |2
;; Luke                 | Vanderhart           |1
```

❶ 0 标记表明为数字补上零（这里是 3 个数字）。

❷ - 标记表明字符串（s）左对齐，最小宽度为 20 个字符。

## 讨论

要在字符串中插入值，有两种不同的选择：可以用 `str`，这种方法很方便，但难以控制值的显示；也可以使用 `format`，它可以精细控制值的显示方法，但需要了解 C 和 Java 风格的格式化字符串。归根结底，采用的工具或复杂程度应该符合当前任务的需求：如果值的默认格式足够，就使用 `str`。如果需要对值的显示有更多控制，就使用 `format`。

### 格式字符串

传给 `format` 的第一个参数就是所谓的格式字符串。这些字符串的文法不是 Clojure 新创或独有的，甚至也不应归功于 Java，它们实际上来自 C 的 `printf` 函数。Clojure 的 `format` 函数使用了 Java 的 `String/format`，它实现了 `printf` 风格的值替换。

格式字符串是一个正常的字符串，其中嵌入了任意数量的格式指定符。格式指定符是一个占位符，稍后将由值取代。最简单的形式是 % 后跟一个类型指定字符。例如，`%d` 是指整数（d 表示数字），`%f` 表示浮点数。除了字符串、整数和浮点数的指定字符，还有字符、日期和不同进制的数字（8 进制和 16 进制）等。

这些格式指定符的特殊之处在于，可以在 % 和类型指定字符之间插入任意多的标记和选项。例如，“`%-10s`”表明提供的字符串（s）应该左对齐（-），总的最小宽度是 10。“`%07.3f`”将一个数变成 0 补齐的数，有 7 个字符宽，包括 3 个小数位（就像杜威十进制系统中使用的数）：

```
(format "%07.3f" 0.005)
;; -> "000.005" ;; 计算机编程、程序和数据书籍的杜威十进制分类
```

要了解格式化字符串的更多内容，请查看 `java.util.Formatter` 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html）。

## 参阅

- 1.3 节“利用部件构建字符串”。

- 1.28 节“利用 clj-time 格式化日期”。

# 1.7 按模式查找字符串

作者：Ryan Neufeld

## 问题

需要测试一个字符串，看看它的组成部分是否符合一个模式。

## 解决方案

要检查字符串中是否存在一个模式，请用 re-find，参数是期望的模式和要检查的字符串。用正则表达式来表示期望的模式（如“foo”或“\d+”）：

```
;; 所有连续的数字
(re-find #"\d+" "I've just finished reading Fahrenheit 451")
;; -> "451"

(re-find #"Bees" "Beads aren't cheap.")
;; -> nil
```

## 讨论

要检查字符串是否包含一个模式，re-find 非常方便。它以正则表达式模式和字符串为参数，返回模式的第一个匹配或 nil。

如果条件更严格，要求整个字符串匹配一个模式，请使用 re-matches。它和 re-find 不一样，不是匹配字符串的任意部分，而是仅匹配整个字符串。

```
;; 在 find 中，#"\w+" 是任何连续的单词字符
(re-find #"\w+" "my-param")
;; -> "my"

;; 但在 matches 中，#"\w+" 意味着 " 全是单词字符 "
(re-matches #"\w+" "my-param")
;; -> nil

(re-matches #"\w+" "justLetters")
;; -> "justLetters"
```

## 参阅

- java.lang.Pattern 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html），其中定义了 Java 支持的正则表达式语法（Clojure 的正则表达式一样）。

- 1.8 节“利用正则表达式从字符串中取出值”，探讨了利用正则表达式从字符串中提取值。
- 1.9 节 “对字符串执行查找和替换”。

# 1.8 利用正则表达式从字符串中取出值

作者：Ryan Neufeld

## 问题

需要提取匹配指定模式的字符串部分。

## 解决方案

使用 `re-seq`，参数是一个正则表达式模式和一个字符串，得到一系列连续的匹配：

```
;; 从句子中提取简单的单词
(re-seq #"\w+" "My Favorite Things")
;; -> ("My" "Favorite" "Things")

;; 提取简单的 7 位电话号码
(re-seq #"\d{3}-\d{4}" "My phone number is 555-1234.")
;; -> ("555-1234")
```

带有匹配组（括号）的正则表达式将返回一个向量，包含所有的完全匹配：

```
;; 提取所有的 Twitter 用户名和 # 标签
(defn mentions [tweet]
  (re-seq #"(@|#)(\w+)" tweet))

(mentions "So long, @earth, and thanks for all the #fish. #goodbyes")
;; -> (["@earth" "@" "earth"] ["#fish" "#" "fish"] ["#goodbyes" "#" "goodbyes"])
```

## 讨论

提供一个简单的模式（没有匹配组的），`re-seq` 会返回一系列的匹配。这是惰性匹配，充分体现了 Clojure 的强大。对一个巨大的字符串调用 `re-seq` 时，不会马上扫描整个字符串，可以增量式地处理这些值，或者将计算工作推迟到应用程序后面的部分。

如果给定的正则表达式包含匹配组，`re-seq` 的做法就有点不一样。不要担心，结果序列仍然是惰性的——不过其值将是向量，而非单调的字符串。向量的第一个值总是完整的匹配，不论是否包含匹配组。后续的值是匹配组括号所捕获的字符串。这些捕获的值将按括号的顺序出现，除非有嵌套的情况。请看下面的例子：

```
;; 利用正则表达式来捕获和分解电话号码及其标题。
(def re-phone-number #"(\w+): \((\d{3})\) (\d{3}-\d{4})")
```

```
(re-seq re-phone-number "Home: (919) 555-1234, Work: (191) 555-1234")
;; -> (["Home: (919) 555-1234" "Home" "919" "555-1234"]
;;     ["Work: (191) 555-1234" "Work" "191" "555-1234"])
```

如果只要在字符串中寻找一次匹配，那就使用 `re-find`。它的行为几乎和 `re-seq` 一样，但只返回第一次匹配的一个值，而不是一系列的匹配值。

除了 `re-seq` 以外，还有另一种方法可以迭代访问字符串中的所有匹配。可以在 `re-matcher` 上反复调用 `re-find`，但我们不建议这样做，因为它不太符合 Clojure 的习惯。改变一个 `re-matcher` 对象，然后反复调用 `re-find` 就是不对，它完全违反了纯函数式的原则。我们强烈建议使用 `re-seq`，而不是 `re-matcher` 和 `re-find`，除非真有好的理由。

## 参阅

- 1.7 节“按模式查找字符串”，查找了字符串中出现的模式。
- 1.9 节“对字符串执行查找和替换”，利用了正则表达式来查找和替换字符串的某些部分。
- `java.lang.Pattern` 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html），其中定义了 Java 支持的正则表达式语法（Clojure 的正则表达式一样）。

# 1.9 对字符串执行查找和替换

作者：Ryan Neufeld

## 问题

需要修改字符串的某些部分，使它们匹配某种确定的模式。

## 解决方案

如果要选择性地替换字符串中的某些部分，多才多艺的 `clojure.string/replace` 就是我们要找的函数。

对于简单的模式，用 `replace` 时带一个普通的字符串，作为它的匹配模式：

```
(def about-me "My favorite color is green!")
(clojure.string/replace about-me "green" "red")
;; -> "My favorite color is red!"

(defn de-canadianize [s]
  (clojure.string/replace s "ou" "o"))
(de-canadianize (str "Those Canadian neighbours have coloured behaviour"
                     " when it comes to word endings"))
;; -> "Those Canadian neighbors have colored behavior when it comes to word
;;     endings"
```

简单的字符串替换只能完成这样的功能。如果需要替代的模式中包含某些可变性，就需要寻找更强大的武器：正则表达式。使用 Clojure 的正则表达式文法（#"..."）来指定正则表达式模式：

```
(defn linkify-comment
  "Add Markdown-style links for any GitHub issue numbers present in comment"
  [repo comment]
  (clojure.string/replace comment
                          #"#(\d+)"
                          (str "[#$1](https://github.com/" repo "/issues/$1)")))

(linkify-comment "next/big-thing" "As soon as we fix #42 and #1337 we
should be set to release!")
;; -> "As soon as we fix
;;     [#42](https://github.com/next/big-thing/issues/42) and
;;     [#1337](https://github.com/next/big-thing/issues/1337) we
;;     should be set to release!"
```

## 讨论

`replace` 是字符串函数中最强大、最复杂的之一。这种复杂性主要来自于它可以进行不同的匹配和替换。

如果传入一个字符串来匹配，`replace` 期待一个字符串来替换。在被查字符串中，所有发生的匹配都会被直接替换成替代字符串。

如果传入一个字符（如 `\c` 或 `\n`）来匹配，`replace` 期待一个字符来替换。就像字符串替换字符串一样，`replace` 的字符替换字符的模式也是直接替换的。

如果传入一个正则表达式来匹配，`replace` 就有趣得多了。正则表达式匹配的一种可能替换是一个字符串，就像在 `linkify-comment` 的例子中那样。这个字符串将特殊的字符组合（如 `$1` 和 `$2`）解释为变量，并替换成匹配结果中的匹配组。在 `linkify-comment` 的例子中，所有数字符号（`#`）后跟着连续的数字（`\d+`）被括号捕获，在替代时作为 `$1` 提供。

如果传入一个正则表达式来匹配，也可以提供一个函数作为替换，而不是一个字符串。在 Clojure 中，如果能传入一个函数作为参数，整个世界都由你作主了。可以在可复用（并且可测试）的函数中捕获替换，根据情况传入不同的函数，甚至传入一个映射表来控制替换：

```
;; linkify-comment 重写，替换时使用独立的函数
(defn linkify [repo [full-match id]]
  (str "[" full-match "](https://github.com/" repo "/issues/" id ")"))

(defn linkify-comment [repo comment]
  (clojure.string/replace comment #"#(\d+)" (partial linkify repo)))
```

如果你以前没用过正则表达式，用一下就会立刻喜欢上它。在修改字符串时，正则表达式

是强大的工具，非常灵活。就像所有强大的新工具那样，很容易被滥用。因为它们简明而紧凑的语法，很容易导致正则表达式既难以解读，又很容易犯错。应该谨慎使用正则表达式，并且只在完全理解其语法时才使用。

要学习和掌握正则表达式的语法，Jeffrey Friedl 的 *Mastering Regular Expressions*, 3rd ed. (O'Reilly) 是一本很好的书。

## 参阅

- 1.7 节“按模式查找字符串”。
- `clojure.string/replace-first`，该函数与 `clojure.string/replace` 几乎一样，但只替换第一次匹配。
- `java.lang.Pattern` 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html），其中定义了 Java 支持的正则表达式语法（Clojure 的正则表达式一样）。

# 1.10　将字符串切分成部分

作者：Ryan Neufeld

## 问题

需要将字符串切分成若干部分。

## 解决方案

使用 `clojure.string/split`，将字符串切分成一组子串。`split` 接受两个参数，一个是待切分的字符串，另一个是切分依据的正则表达式：

```
(clojure.string/split "HEADER1,HEADER2,HEADER3" #",")
;; -> ["HEADER1" "HEADER2" "HEADER3"]

(clojure.string/split "Spaces   Newlines\n\n" #"\s+")
;; -> ["Spaces" "Newlines"]
```

## 讨论

除了简单地按正则表达式切分之外，`split` 允许你控制切分的次数。你可以利用可选的 `limit` 参数做到这一点。`limit` 最明显的效果就是限制结果集合里值的个数。也就是说，`limit` 并非总像你期望的那样工作，而且即使不提供这个参数，也是有意义的。

没有 `limit` 时，`split` 函数将返回所有可能的切分，但排除尾部的空匹配：

```
;; 按空白字符切分，没有指明 limit，执行了隐式的去除空白字符操作
```

```
(clojure.string/split "field1     field2 field3  " #"\s+")
;; -> ["field1" "field2" "field3"]
```

如果你想要的就是所有匹配，包括尾部的空匹配，那么可以将 `limit` 指定为 `-1`。

```
;; 在 CSV 解析时，行末的空匹配仍然是有意义的
(clojure.string/split "ryan,neufeld," #"," -1)
;; -> ["ryan" "neufeld" ""]
```

将 `limit` 指定为其他正数，导致 `split` 最多返回 `limit` 个子串。

```
(def data-delimiters #"[ :-]")

;; 没有 limit，按所有定界符切分
(clojure.string/split "2013-04-05 14:39" data-delimiters)
;; -> ["2013" "04" "05" "14" "39"]

;; Limit 为 1，返回包含这个字符串的集合
(clojure.string/split "2013-04-05 14:39" data-delimiters 1)
;; -> ["2013-04-05 14:39"]

;; Limit 为 2
(clojure.string/split "2013-04-05 14:39" data-delimiters 2)
;; -> ["2013" "04-05 14:39"]

;; Limit 为 100
(clojure.string/split "2013-04-05 14:39" data-delimiters 100)
;; -> ["2013" "04" "05" "14" "39"]
```

## 参阅

- `clojure.string` 命名空间 API 文档（http://clojure.github.io/clojure/clojure.string-api.html）。
- 1.7 节“按模式查找字符串”。
- 1.8 节“利用正则表达式从字符串中取出值”。

# 1.11 基于数量为字符串加复数

作者：Ryan Neufeld

## 问题

需要基于数量为单词添加复数，如“0 eggs”或“1 chicken”。

## 解决方案

如果需要得到 Ruby on Rails 风格的复数，请使用 Roman Scherer 的 `inflections` 库（https://github.com/r0man/inflections-clj）。

要继续这个实例，先用 lein-try 开始 REPL[3]：

```
$ lein try inflections
```

使用 inflections.core/pluralize 时带一个计数参数，如果计数不为 1，尝试给出单词的复数：

```
(require '[inflections.core :as inf])

(inf/pluralize 1 "monkey")
;; -> "1 monkey"

(inf/pluralize 12 "monkey")
;; -> "12 monkeys"
```

如果有特殊或非标准的复数形式，可以为 pluralize 提供第 3 个可选参数，指定自己的复数形式：

```
(inf/pluralize 1 "box" "boxen")
;; -> "1 box"

(inf/pluralize 3 "box" "boxen")
;; -> "3 boxen"
```

## 讨论

对于展现给用户的文字，变形是关键。让程序或网站的输出人性化，有益于建立值得信任的专业形象。对于用户友好的、人性化的文字，Ruby on Rails（http://rubyonrails.org/）用它的 Active Support::Inflections 类，建立了一个黄金标准。Inflections#pluralize 就是这样一种变形，但 Inflections 充满了听起来讨人喜爱的方法，以“ize”结尾，改变字符串的形态。在 Clojure 环境中，Inflections 几乎提供了所有这些功能。

Inflections 库中有两个有趣的函数，plural 和 singular。这两个函数有点像单词复数的 upper-case 和 lower-case。plural 将单词转变成复数形式，singular 将单词转换成单数形式。这些转换基于 inflections.plural 中的一些规则。你可以利用 inflections.core/plural，添加自己的复数规则！

```
(inf/plural "box")
;; -> "boxes"

;; 以 'ox' 结尾的单词复数加 'en'（而不是 'es'）
(inf/plural! #"(ox)(?i)$" "$1en")

(inf/plural "box")
;; -> "boxen"
```

注 3：如果还没有安装 lein-try，请按照前言中“我们的金童 lein-try”的指令安装。

```
;; plural 也是 pluralize 的基础…
(inf/pluralize 2 "box")
;; -> "2 boxen"
```

该库也支持 camelize、parameterize 和 ordinalize 等变形：

```
;; 将 "snake_case" 变为 "CamelCase"
(inf/camelize "my_object")
;; -> "MyObject"

;; 清理字符串，用于 URL 参数
(inf/parameterize "My most favorite URL!")
;; -> "my-most-favorite-url"

;; 将基数变为序数
(inf/ordinalize 42)
;; -> "42nd"
```

## 参阅

- inflections-clj GitHub 库（https://github.com/r0man/inflections-clj/），了解最新支持的变形清单。

# 1.12　在字符串、符号和关键字之间转换

作者：Colin Jones

## 问题

有一个字符串、符号或关键字，需要转换成不同的、像字符串一样的数据类型。

## 解决方案

要将字符串转换成符号，请使用 symbol 函数：

```
(symbol "valid?")
;; -> valid?
```

要将符号转换成字符串，请使用 str：

```
(str 'valid?)
;; -> "valid?"
```

如果有一个关键字要转换成字符串，可以使用 name，如果需要头上的冒号，可以使用 str：

```
(name :triumph)
```

```
;; -> "triumph"

;; 或者，包含头上的冒号
(str :triumph)
;; -> ":triumph"
```

要将符号或字符串转换成关键字，请使用 keyword：

```
(keyword "fantastic")
;; -> :fantastic

(keyword 'fantastic)
;; -> :fantastic
```

要将关键字转换成符号，需要一个中间步骤，即通过 name：

```
(symbol (name :wonderful))
;; -> wonderful
```

## 讨论

这里主要的转换函数是 str、keyword 和 symbol，每个名称对应它返回的数据类型。其中 symbol 有一点严格，它允许的参数只能是字符串，这就是为什么关键字转换成符号需要一个中间步骤。

这些类型之间还有另一个不同之处：即关键字和符号可能带有命名空间，中间用斜杠（/）分开。对于这种类型的关键字和符号，name 函数可能够用，也可能不够用，这取决于使用的情况：

```
;; 如果只想要关键字的名字部分
(name :user/valid?)
;; -> "valid?"

;; 如果只想要命名空间
(namespace :user/valid?)
;; -> "user"
```

通常，实际上想要的是两个部分。可以分别取得这两个部分，然后将它们连接起来，中间加上 /，但还有更容易的方式。Java 有大量高效率的方法，来处理不可修改的字符串。可以利用 java.lang.String.substring(int)，消除冒号开头的字符串的第一个字符：

```
(str :user/valid?)
;; -> ":user/valid?"

(.substring (str :user/valid?) 1)
;; -> "user/valid?"
```

关于字符串的更多方法，参见 java.lang.String 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/lang/String.html）。

你可以容易地将带命名空间的符号转换成关键字，就像不带命名空间的符号一样，但再次强调，反方向转换（关键字转换成符号）需要增加一个步骤：

```
(keyword 'produce/onions)
;; -> :produce/onions

(symbol (.substring (str :produce/onions) 1))
;; -> produce/onions
```

最后，`keyword` 和 `symbol` 函数都有两个参数的版本，允许你分别传入命名空间和名字。有时候这样更好，例如，如果两个值中有一个或两个已经定义在 `def`、`let` 或其他范围中：

```
(def shopping-area "bakery")

(keyword shopping-area "bagels")
;; -> :bakery/bagels

(symbol shopping-area "cakes")
;; -> bakery/cakes
```

这三个类似字符串的数据类型在不同情况下分别适用，如何选择又是另外一个话题。但常常需要在它们之间进行转换，所以，将 `keyword`、`symbol`、`str`、`namespace` 和 `name` 放入你的工具栏中会很方便。

## 参阅

- 1.5 节“字符与整数的转换”。

# 1.13　利用非常大或非常小的数来保持精度

作者：Ryan Neufeld

## 问题

需要精确地处理数字，尤其是那些非常大或非常小的数字，消除 `double` 这样的浮点表示法所隐含的不精确。

## 解决方案

首先，要知道 Clojure 支持以指数形式表示数字，允许简洁地表示非常大或非常小的数字：

```
;; 阿伏伽德罗常数
6.0221413e23
;; -> 6.0221413E23

;; 1 埃相当于多少米
1e-10
```

```
;; -> 1.0E-10
```

如果整数值超过了有边界类型（如 long）的上界，就会导致整数溢出错误。使用“引号”版本的数字操作，如 - 或 *，可以提升到 Big 类型：

```
(* 9999 9999 9999 9999 9999)
;; ArithmeticException integer overflow  clojure.lang.Numbers.throwIntOverflow

(*' 9999 9999 9999 9999 9999)
;; -> 99950009999000049999N
```

## 讨论

Clojure 有一些数值类型：int 和 long、double、BigInteger 以及 BigDecimal。有边界的类型（int、long 和 double）都会在这些类型的总边界内无缝转换。超出这个边界会导致两种情况发生：对于整数，会产生整数溢出错误；对于浮点数，结果将变成无穷大。在使用整数时，可以用引号版本的 +、-、* 和 / 来避免这个错误。这些操作支持任意的精度，在需要时会将整数提升为 BigInteger。

浮点值更难处理一些。引号版本的数值操作没有帮助。你需要使用 BigDecimal 类型，来“传染”操作。在 Clojure 中，BigInteger 和 BigDecimal 就是所谓的“传染”类型。只要在操作中引入一个“大”数，它就会传染所有接下来的结果。你可以通过某种操作，例如用 BigDecimal 的 1 去乘一个数，但更简单的方法是使用 bigdec 或 bigint，手工提升一个值：

```
(* 2 Double/MAX_VALUE)
;; -> Double/POSITIVE_INFINITY

(* 2 (bigdec Double/MAX_VALUE))
;; -> 3.5953862697246314E+308M
```

传染不只是发生在这些 Big 类型中，它也发生在整数到浮点数的边界上。对于整数来说，浮点数是传染的。包含任何浮点值的运算都会得到浮点值。

## 参阅

- 1.14 节“使用有理数”，探讨了在使用有理数时保持精度。

# 1.14　使用有理数

作者：Ryan Neufeld

## 问题

需要以绝对的精度操作有理数。

## 解决方案

在操作整数（或其他有理数）时，你可能希望保持精度，包括循环有理数，如 1/3（0.333…）：

```
(/ 1 3)
;; -> 1/3

(type (/ 1 3))
;; -> clojure.lang.Ratio

(* 3 (/ 1 3))
;; -> 1N
```

在 double 类型上使用 rationalize，将它们强制转换成有理数，以避免损失精度：

```
(+ (/ 1 3) 0.3)
;; -> 0.6333333333333333

(rationalize 0.3)
;; -> 3/10

(+ (/ 1 3) (rationalize 0.3))
;; -> 19/30
```

## 讨论

在处理数值时，Clojure 尽可能保持精度，尤其是对整数。在整数除法中，Clojure 将商表示为精确的整数比，而不是有损耗的 double 类型，从而保持准确性。但这种准确性是有代价的，和其他简单类型的操作相比，有理数的操作要慢得多。正如 1.13 节“利用非常大或非常小的数来保持精度”中讨论的那样，精度总是性能的妥协，需要根据面对的问题来衡量。

在同时操作 double 和有理数时，要小心。由于 Clojure 的类型传染方式，对两个类型执行操作会导致有理数被强制转换成 double 类型。这种转换对单个操作不一定是不准确的，但类型的改变可能导致不准确性悄悄发生。

在处理 double 时，要保持准确性，请使用 rationalize 函数。这个函数返回任何数的有理数值。对可能是 double 类型的值调用 rationalize，能保持绝对精确（但要以牺牲性能为代价）。

## 参阅

- 1.13 节“利用非常大或非常小的数来保持精度”。

# 1.15 解析数字

作者：Ryan Neufeld

## 问题

需要从字符串中解析出数。

## 解决方案

对于“正常”大小的整数或双精度数，请使用 `Integer/parseInt` 或 `Double/parseDouble` 来解析：

```
(Integer/parseInt "-42")
;; -> -42

(Double/parseDouble "3.14")
;; -> 3.14
```

## 讨论

什么是“正常”大小的数？对于 `Integer/parseInt` 来说，正常就是小于 `Integer/MAX_VALUE`（2147483647）。对于 `Double/parseDouble` 来说，正常就是小于 `Double/MAX_VALUE`（约 1.79 × 10^308）。

如果要解析的数要么太大，要么精度太高，就需要使用 `BigInteger` 或 `BigDecimal`，以避免精度损失。

多才多艺的 `bigint` 和 `bigdec` 函数可以将字符串（或任何其他数字类型）强制转换成无限精度的容器：

```
(bigdec "3.1415926535897932384626433832795028841971")
;; -> 3.1415926535897932384626433832795028841971M

(bigint "122333444455555666666777777788888888999999999")
;; -> 122333444455555666666777777788888888999999999N
```

## 参阅

- `Integer/parseInt`（http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html#parseInt(java.lang.String)）和 `Double/parseDouble`（http://docs.oracle.com/javase/7/docs/api/java/lang/Double.html#parseDouble(java.lang.String)）的 API 文档。

# 1.16 数的截断和舍入

作者：Ryan Neufeld

## 问题

需要截断或舍入小数，成为低精度的数。

## 解决方案

如果只关心整数部分，请使用 `int`，将该数强制转换成整数。当然，这会完全丢弃小数部分，没有任何的进位：

```
(int 2.0001)
;; -> 2

(int 2.999999999)
;; -> 2
```

如果仍然期望某种程度的精度，可能希望舍入。可以使用 `Math/round`，进行简单的舍入：

```
(Math/round 2.0001)
;; -> 2

(Math/round 2.999)
;; -> 3

;; 这等价于：
(int (+ 2.99 0.5))
;; -> 3
```

如果希望进行不平衡的舍入，诸如无条件地“舍入到较大数”或“舍入到较小数”，那就应该分别使用 `Math/ceil` 或 `Math/floor`：

```
(Math/ceil 2.0001)
;; -> 3.0

(Math/floor 2.999)
;; -> 2.0
```

你会注意到这些函数返回小数。要得到整数，请在 `ceil` 或 `floor` 外面调用 `int`。

## 讨论

“舍入”数字最简单的方法就是截断。`int` 会完成这项任务，将浮点数强制转换成整数，简单地将小数部分丢弃。这在数学上不一定对，但如果面对的问题允许，这样肯定很方便。

Math/round 是更高一级的舍入技术。就像 Clojure 中的许多其他原生操作函数一样，这门语言倾向于“不重新发明轮子”。Math/ round 是一个 Java 函数，舍入的方法是先加上 1/2，再像 int 那样丢弃小数部分。

对于更高级的舍入，诸如控制小数位或复杂的进位模式，可能需要使用 with-precision 函数。你也许已经知道 BigDecimal 类型背后是 Java 类，但也许不知道 Java 提供了一些控制方法来调整 BigDecimal 的计算。with-precision 提供了这些方法。

with-precision 是一个宏，它接受一个 BigDecimal 的精度模式和任意数目的表达式，在 BigDecimal 的上下文中执行这些表达式，调整到那种精度。那么精度看起来是怎样的？好吧，它有点奇怪。最基本的精度就是一个正整数“范围”值。这个值指定了小数点后保留几位。更复杂的精度涉及一个 :rounding 值，以键 / 值对的形式指定，如 :rounding FLOOR（这当然是一个宏，为什么不呢？）。如果未指定，舍入模式就是 HALF_UP，但可以是 CEILING、FLOOR、HALF_UP、HALF_DOWN、HALF_EVEN、UP、DOWN、UNNECESSARY 中的任何一个。

```
(with-precision 3 (/ 7M 9))
;; -> 0.778M

(with-precision 1 (/ 7M 9))
;; -> 0.8M

(with-precision 1 :rounding FLOOR (/ 7M 9))
;; -> 0.7M
```

关于 with-precision，有一个“坑”值得一提：它只改变 BigDecimal 运算的行为，其他常规的运算不变。必须用字面值（3M）在表达式中引入 BigDecimal 的值，或者使用 bigdec 函数：

```
(with-precision 3 (/ 1 3))
;; -> 1/3

(with-precision 3 (/ (bigdec 1) 3))
;; -> 0.333M
```

## 参阅

- 1.13 节“利用非常大或非常小的数来保持精度”，探讨了 BigDecimal 的更多内容，尤其是类型传染。
- 1.17 节“模糊比较”。

# 1.17 模糊比较

作者：Ryan Neufeld

## 问题

需要检测是否相等，并容忍一些小误差。这在比较浮点数时尤其是个问题，因为浮点数容易在反复操作之后发生“漂移”。

## 解决方案

Clojure 没有内建的函数实现容错相等比较，或者说“模糊比较”，大家常这么说。因为很容易实现你自己的 `fuzzy=` 函数：

```
(defn fuzzy= [tolerance x y]
  (let [diff (Math/abs (- x y))]
    (< diff tolerance)))

(fuzzy= 0.01 10 10.000000000001)
;; -> true

(fuzzy= 0.01 10 10.1)
;; -> false
```

## 讨论

`fuzzy=` 的工作方式就像大多数模糊比较算法一样：首先找到两个操作数的绝对差值，其次测试差值是否小于给定的容忍值。当然，没有什么规定容忍值必须是某个很小的小数。如果要比较很大的数，希望忽略 1000 以下的差异，也可以把容忍值设为 `1000`。

即使使用 `fuzzy=`，在比较浮点值时也需要小心，尤其是那些差值很接近容忍值的数。当差值接近提供的容忍值时，你可能发现结果有点奇怪：

```
(- 0.22 0.23)
;; -> -0.010000000000000009

(- 0.23 0.24)
;; -> -0.009999999999999981
```

虽然这很奇怪，但也在预料之中。IEEE 754 规范是关于浮点数的，它有意限制了格式，这是精度和性能的折中。如果想要绝对的精确，就应该使用 `BigDecimal` 或 `BigInt`。参见 1.13 节“利用非常大或非常小的数来保持精度”，该节探讨了这两个类型的更多内容。

`fuzzy=` 函数的这种写法，有一些有趣的副作用。首先，使用容忍值作为第一个参数，让它能够利用 `partial` 实现部分应用的相等判断函数，指定某个容忍值：

```
(def equal-within-ten? (partial fuzzy= 10))

(equal-within-ten? 100 109)
;; -> true

(equal-within-ten? 100 110)
;; -> false
```

如果想在排序中使用模糊比较怎么办？ sort 函数接受一个可选的参数作为判断或比较方法。让我们来写一个 fuzzy-comparator 函数，它按给定的容忍值返回一个比较方法：

```
(defn fuzzy-comparator [tolerance]
  (fn [x y]
    (if (fuzzy= tolerance x y) ; ❶
      0
      (compare x y)))) ; ❷

(sort (fuzzy-comparator 10) [100 11 150 10 9])
;; -> (11 10 9 100 150) ; 100和150移动了位置,但11、10和9没有
```

❶ 如果两个比较的值在 tolerance 范围之内，返回 0 表示它们相等。

❷ 否则退回到正常的 compare。

## 参阅

- 关于 IEEE 浮点数的维基百科（http://en.wikipedia.org/wiki/IEEE_floating_point）。
- 1.13 节“利用非常大或非常小的数来保持精度”。
- 1.16 节“数的截断和舍入”。

# 1.18　三角计算

作者：Ryan Neufeld

## 问题

需要实现一些数学函数，要求三角计算。

## 解决方案

所有的三角函数都可以通过 java.lang.Math（http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html）来访问，它以 Math 的形式提供。使用它们就像其他带命名空间的函数一样：

```
;; 计算 sin(a + b). 公式是
;; sin(a + b) = sin a * cos b + sin b cos a
```

```
(defn sin-plus [a b]
  (+ (* (Math/sin a) (Math/cos b))
     (* (Math/sin b) (Math/cos a))))

(sin-plus 0.1 0.3)
;; -> 0.38941834230865047
```

三角函数操作的值是以弧度表示的。如果值是以角度表示的，如经度或纬度，就需要先将它转换成弧度。使用 `Math/toRadians` 将角度转换成弧度：

```
;; 计算地球上两点间的公里数
(def earth-radius 6371.009)

(defn degrees->radians [point]
  (mapv #(Math/toRadians %) point))

(defn distance-between
"Calculate the distance in km between two points on Earth. Each
 point is a pair of degrees latitude and longitude, in that order."
([p1 p2] (distance-between p1 p2 earth-radius))
([p1 p2 radius]
  (let [[lat1 long1] (degrees->radians p1)
        [lat2 long2] (degrees->radians p2)]
    (* radius
       (Math/acos (+ (* (Math/sin lat1) (Math/sin lat2))
                     (* (Math/cos lat1)
                        (Math/cos lat2)
                        (Math/cos (- long1 long2)))))))))

(distance-between [49.2000 -98.1000] [35.9939, -78.8989])
;; -> 2139.42827188432
```

## 讨论

有些人可能吃惊于 Clojure 没有自己的内部数学命名空间，但是为什么要“重新发明轮子”呢？尽管 Java 的名声不太好，但它也可以有效率，尤其是在数学方面。Clojure 与 Java 互操作的形式加上它的语法糖，使得利用 `java.lang.Math` 来完成数学运算变得非常愉快。

`java.lang.Math` 不只有三角运算。它也包含一些有用的函数，进行指数、对数和开根号运算。`java.lang.Math` 的 javadoc 中（http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html）有完整的清单。

## 参阅

- 8.5 节“利用类型暗示减轻性能问题”，探讨了改进性能的小窍门。

# 1.19 根据不同的进制输入和输出整数

作者：Ryan Neufeld

## 问题

需要以不同的进制（如十六进制或二进制），在 Clojure REPL 或代码中输入一些数。

## 解决方案

要指定常数的进制，就是在它前面加上进制数（如 2、16 等）和字母 `r`。从 2 到 36 的进制都是有效的（当然，可以用 10 个数字和 26 个字母）：

```
2r101010
;; -> 42

3r1120
;; -> 42

16r2A
;; -> 42

36rABUNCH
;; -> 624567473
```

要输出整数，请用 Java 方法 `Integer/toString`：

```
(Integer/toString 13 2)
;; -> "1101"

(Integer/toString 42 16)
;; -> "2a"

(Integer/toString 35 36)
;; -> "z"
```

## 讨论

不像大多数 Clojure 函数的次序，这个方法先接受一个整数参数，然后再是可选的进制。这使得它很难部分地应用，除非包装成另一个函数。你可以为 `Integer/toString` 写一个小包装函数，来实现这一点：

```
(defn to-base [radix n]
  (Integer/toString n radix))

(def base-two (partial to-base 2))

(base-two 9001)
;; -> "10001100101001"
```

## 参阅

- 1.6 节“格式化字符串”，探讨了关于 format 的信息（o 和 x 分别指定用八进制和十六进制打印整数）。
- 1.15 节“解析数字”。

# 1.20　计算数值集合的统计值

作者：Ryan Neufeld 和 Jean Niklas L’orange

## 问题

需要计算数值集合的简单统计值，如均值、中位数、众数和标准差。

## 解决方案

要计算集合的平均值，就用总和除以集合的元素个数 count：

```
(defn mean [coll]
  (let [sum (apply + coll)
        count (count coll)]
    (if (pos? count)
      (/ sum count)
      0)))

(mean [1 2 3 4])
;; -> 5/2

(mean [1 1.6 7.4 10])
;; -> 5.0

(mean [])
;; -> 0
```

要找出集合的中位数，就是对它的值排序，并取得中间的值。当然，对集合有偶数个元素的情况有特殊的考虑。在这种情况下，中位数是两个中间值的平均值：

```
(defn median [coll]
  (let [sorted (sort coll)
        cnt (count sorted)
        halfway (int (/ cnt 2))]
    (if (odd? cnt)
      (nth sorted halfway) ; ❶
      (let [bottom (dec halfway)
            bottom-val (nth sorted bottom)
            top-val (nth sorted halfway)]
        (mean [bottom-val top-val])))))) ; ❷
```

```
(median [5 2 4 1 3])
;; -> 3

(median [7 0 2 3])
;; -> 5/2 ; The average of 2 and 3.
```

❶ 在 `coll` 有奇数个元素的情况下，只要用 `nth` 取得那个元素。

❷ 如果 `coll` 有偶数个元素，找到另一个中间值的下标（`bottom`），然后取 top 和 bottom 的平均值。

要找出集合的众数（最常出现的值），就要用 `frequencies` 记录元素的出现次数。然后处理这些记录，取得众数的离散列表：

```
(defn mode [coll]
  (let [freqs (frequencies coll)
        occurrences (group-by second freqs)
        modes (last (sort occurrences))
        modes (->> modes
                   second
                   (map first))]
     modes))

(mode [:alan :bob :alan :greg])
;; -> (:alan)

(mode [:smith :carpenter :doe :smith :doe])
;; -> (:smith :doe)
```

### 标准差

要找出样本的*标准差*，就要完成以下步骤。

(1) 对集合中的每个值，减去平均值 `mean`，结果再平方。

(2) 然后，将这些值相加。

(3) 将结果除以值的个数减 1。

(4) 最后，对前面的结果取平方根：

```
(defn standard-deviation [coll]
  (let [avg (mean coll)
        squares (for [x coll]
                  (let [x-avg (- x avg)]
                    (* x-avg x-avg)))
        total (count coll)]
    (-> (/ (apply + squares)
           (- total 1))
        (Math/sqrt))))

(standard-deviation [4 5 2 9 5 7 4 5 4])
;; -> 2.0
```

```
(standard-deviation [4 5 5 4 4 2 2 6])
;; -> 1.4142135623730951
```

## 讨论

mean 和 median 在 Clojure 中都很容易实现，但 mode 需要一点工作量。mode 与 mean 或 median 有点不同，它通常对非数值数据有意义。计算集合的众数比较复杂一点，和相关的一些数值函数相比，它基本上需要大量的处理。

下面是 mode 工作方式的分解。

```
(defn mode [coll]
  (let [freqs (frequencies coll)           ; ❶
        occurrences (group-by second freqs) ; ❷
        modes (last (sort occurrences))     ; ❸
        modes (->> modes                    ; ❹
                   second
                   (map first))]
     modes))
```

❶ frequencies 返回一个映射表，记录了 coll 中每个元素的出现次数。它的样子可能是 {:a 1 :b 2}。

❷ group-by 带上 second 将 freqs 映射表倒转，键变成值，并将重复的分到一个组里。这将 {:a 1 :b 1} 变成了 {1 [[:a 1] [:b 1]]}。

❸ 出现次数列表现在可以排序了。排好序的列表中，最后一对就是众数，或最常出现的值。

❹ 最后一步是将原始的众数对处理成离散的值，利用 second 将 [2 [[:alan 2]]] 变成 [[:alan 2]]，然后用 (map first) 将它变成 (:alan)。

标准差用来测量在平均的情况下，一组数据的单个值偏离平均值的程度。标准差越高，单个值就离得越远（平均来说）。standard-deviation 比 mean、median 和 mode 更数学化。请一步一步地跟着这个函数执行。

```
(defn standard-deviation [coll]
  (let [avg (mean coll)                 ; ❶
        squares (for [x coll]           ; ❷
                  (let [x-avg (- x avg)]
                    (* x-avg x-avg)))
        total (count coll)]
    (-> (/ (apply + squares)            ; ❸
        (- total 1))
     (Math/sqrt))))
```

❶ 计算集合的平均值。

❷ 对于每个值，计算该值与平均值之差的平方。

❸ 最后，将这些平方加起来，除以元素个数减 1，再开平方，得到样本的标准差。

如果有完整的集合，就可以计算总体标准差，除以 `total` 而不是 `(- total 1)`。

## 参阅

- 关于标准差及其用途，可以参见维基百科关于标准差的文章（http://en.wikipedia.org/wiki/Standard_deviation）。

# 1.21 位操作

作者：Ryan Neufeld

## 问题

需要对数字进行位操作。

## 解决方案

与 C 或 C++ 这样的系统语言相比，位操作在高级语言（如 Clojure）中不太常见，但在那些系统语言中学到的技术仍然有用。Clojure 在它的核心命名空间中提供了一些位操作，前缀是 `bit-`。位操作在一种情况下非常有用，那就是将大量二进制标识压缩到一个值中：

```
;; 将 Unix 文件系统标识的子集放到一个整数中
(def fs-flags [:owner-read :owner-write
              :group-read :group-write
              :global-read :global-write])

;; 将标识放到 " 标识 -> 位 " 的映射表中
(def bitmap (zipmap fs-flags
                    (map (partial bit-shift-left 1) (range))))
;; -> {:owner-read 1, :owner-write 2, :group-read 4, ...}

(defn permissions-int [& flags]
  (reduce bit-or 0 (map bitmap flags)))

(def owner-only (permissions-int :owner-read :owner-write))
(Integer/toBinaryString owner-only)
;; -> "11"

(def read-only (permissions-int :owner-read :group-read :global-read))
(Integer/toBinaryString read-only)
;; -> "10101"
```

```
(defn able-to? [permissions flag]
  (not= 0 (bit-and permissions (bitmap flag))))

(able-to? read-only :global-read)  ;; -> true
(able-to? read-only :global-write) ;; -> false
```

## 讨论

Clojure 在它的核心库中提供了完整的位操作。这包括了逻辑操作 and 和 or，它们的取反，以及移位等。在使用位操作时，常常需要查看整数的二进制表示。Java 的 `Integer/toBinaryString` 可以方便地打印出数字的二进制表示。

很有趣的是，核心库中也包括 `bit-set` 和 `bit-test`。这两个操作用于集合或测试整数中的单个位。不像 `bit-and` 那样需要操作两组多个位，你可以操作感兴趣的标识的下标。这极大地简化了前面的例子：

```
;; 将 Unix 文件系统标识的子集放到一个整数中
(def fs-flags [:owner-read :owner-write
              :group-read :group-write
              :global-read :global-write])

(def bitmap (zipmap fs-flags
                    (map #(.indexOf fs-flags %) fs-flags)))

(def no-permissions 0)
(def owner-read (bit-set no-permissions (:owner-read bitmap)))

(Integer/toBinaryString owner-read)
;; -> "1"

;; 分配全局许可……
(def anything (reduce #(bit-set %1 (bitmap %2)) no-permissions fs-flags))
(Integer/toBinaryString anything)
;; -> "111111"
```

## 参阅

- 8.6 节“用原生 Java 数组进行快速数学运算”。

# 1.22　生成随机数

作者：Ryan Neufeld

## 问题

需要生成随机数。

## 解决方案

Clojure 提供了一些伪随机数生成函数。

要生成 `0.0` 到 `1.0`（但不包含）的随机浮点数，请使用 `rand`：

```
(rand)
;; -> 0.0249306187447903

(rand)
;; -> 0.9242089829055088
```

要生成随机整数，请使用 `rand-int`：

```
;; 模拟 6 面的骰子
(defn roll-d6 []
  (inc (rand-int 6)))

(roll-d6)
;; -> 1

(roll-d6)
;; -> 3
```

## 讨论

除了生成 `0.0` 到 `1.0` 的数之外，`rand` 也接受一个可选参数，指定一个不包含在内的最大值。例如 `(rand 5)` 将返回一个浮点数，范围从 `0.0`（包含）到 `5.0`（不包含）。

`(rand-int 5)` 则返回一个随机整数，从 `0`（包含）到 `5`（不包含）。乍一看，`rand-int` 似乎很适合用来从向量或列表中选择一个随机元素。但这太麻烦了。请使用 `rand-nth`，从任意的序列集合（也就是能响应 `nth` 的集合）中取得一个随机元素：

```
(rand-nth [1 2 3])
;; -> 1

(rand-nth '(:a :b :c))
;; -> :c
```

但这对 set 或 hash map 不起作用。如果希望从 set 这样的非序列集合中取得一个随机元素，请先使用 `seq` 将该集合变成一个序列，再调用 `rand-nth`：

```
(rand-nth (seq #{:heads :tails}))
;; -> :heads
```

如果希望随机排列一个集合，请使用 `shuffle`，得到集合的随机排列：

```
(shuffle [1 2 3 4 5 6])
;; -> [3 1 4 5 2 6]
```

## 参阅

- `java.util.Random` 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/util/Random.html）。
- 10.3 节“通过随机输入进行彻底测试”。

# 1.23 操作货币

作者：Ryan Neufeld

## 问题

需要操作一些代表货币的值。

## 解决方案

使用 Money 库（https://github.com/clojurewerkz/money）来表示、操作和存储带货币单位的值。

要继续这个实例，请将 `[clojurewerkz/money "1.4.0"]` 加入项目依赖关系中，或用 `lein-try` 开始 REPL：

```
$ lein try clojurewerkz/money
```

`clojurewerkz.money.amounts` 命名空间包含了创建、修改、比较货币单位的函数：

```
(require '[clojurewerkz.money.amounts    :as ma])
(require '[clojurewerkz.money.currencies :as mc])

;; $2.00 in USD
(def two (ma/amount-of mc/USD 2))
two
;; -> #<Money USD 2.00>



(ma/plus two two)
;; -> #<Money USD 4.00>

(ma/minus two two)
;; -> #<Money USD 0.00>

(ma/< two (ma/amount-of mc/USD 2.01))
;; -> true

(ma/total [two two two two])
;; -> #<Money USD 8.00>
```

## 讨论

操作货币是严肃的事情。处理货币时，千万不要信任内建的数值类型，尤其是浮点类型。这些类型本来就不是用来记录和操作货币的，不能提供要求的语义和精度。特别是 IEEE 754 标准的浮点值在设计时就包含了一定的不精确性：

```
(- 0.23 0.24)
;; -> -0.009999999999999981
```

应该坚持使用专门为处理货币而定制的库。Money 库封装了值得信任的、经过实战检验的 Java 库 Joda-Money。除了运算外，Money 还提供了大量功能，包括舍入和货币转换：

```
(ma/round (ma/amount-of mc/USD 3.14) 0 :down)
;; -> #<Money USD 3.00>

(ma/convert-to (ma/amount-of mc/CAD 152.34) mc/USD 1.01696 :down)
;; -> #<Money USD 154.92>
```

`round` 函数接受 4 个参数。前 3 个是货币的数量、缩放因子和舍入模式。缩放因子是个有点奇怪的参数。如果你曾用 `BigDecimal` 做过缩放，可能对此有点熟悉，它也有一样的缩放因子。缩放因子为 -1 表示要放大 10 倍，0 表示 1 倍，等等。进一步的细节，可以查看 Joda-Money 库中 `Money` 类的 `rounded`（http://joda-money.sourceforge.net/apidocs/src-html/org/joda/money/Money.html#line.1173）方法的 javadoc 文档。最后一个参数是舍入模式，有几种可以选择。`:ceiling` 和 `:floor` 舍入的方向是正无穷或负无穷，`:up` 和 `:down` 舍入的方向是靠近或远离 0。最后，`:half-up`、`:half-down` 和 `:half-even` 舍入的方向是最近的相邻数，分别是向上、向下或等距离时取偶数相邻数。

`clojurewerkz.money.amounts/convert-to` 是简单得多的函数。`convert-to` 接受货币数量、目标货币、转换因子和舍入模式。Money 没有提供自己的转换因子，因为汇率经常改变，所以需要寻找声誉好的汇率数据来源。不幸的是，这一点我们帮不上忙。

Money 也支持一些不同的持久和序列化中介，包括 Cheshire（https://github.com/dakrone/cheshire），实现与 JSON 的相互转换，也包括 Monger（http://clojuremongodb.info/），将货币值持久到 MongoDB。

## 参阅

- 1.13 节“利用非常大或非常小的数来保持精度”，以及 1.16 节“数的截断和舍入”。

# 1.24 生成唯一ID

作者：Ryan Neufeld

## 问题

需要生成唯一 ID。

## 解决方案

使用 Java 的 `java.util.UUID/randomUUID` 来生成通用唯一 ID（UUID）：

```
(java.util.UUID/randomUUID)
;; -> #uuid "5358e6e3-7f81-40f0-84e5-750e29e6ee05"

(java.util.UUID/randomUUID)
;; -> #uuid "a6f92a6f-f736-468f-9e26-f392852825f4"
```

## 讨论

在构建系统时，常常需要为对象或记录指定唯一的 ID。ID 通常是简单的整数，随时间单调增长。但这样做并非没有问题。

这样做不能混合不同来源的对象的 ID，更糟的是，它们揭示了数据的数量和输入量。

这时就需要 UUID。UUID 即通用唯一标识符，它是 128 位的随机数字，几乎在整个宇宙中都是唯一的。当然，这有点言过其实。请参考 RFC 4122（http://www.ietf.org/rfc/rfc4122.txt），了解 UUID 的更详细信息：它们如何生成以及背后的数学原理。

你可能注意到，Clojure 在打印 UUID 时，前面有 `#uuid`。这是读取程序的字面值标签。它作为一种捷径，让 Clojure 的读取程序读取并初始化 UUID 对象。读取程序字面值很像字符串字面值或数字字面值，如 `"Hi"` 或 `42`，但它们能记录更复杂的数据类型。

这使得 edn（https://github.com/edn-format/edn，可扩展数据表示法）这样的格式能够以共同的术语沟通、交换 UUID 这样的对象，而不用求助于字符串驻留（string interning）以及相应的定制解析逻辑。

### 顺序 ID

从顺序 ID 改为 UUID，会丧失一种可能，即根据随时间推移而增长的数字来排序的可能。如果生成的 UUID 既唯一又可排序，那会怎样？ Datomic 用它的 `datomic.api/squuid` 函数，做了类似的事情。

Datomic 的 `squuid` 近似切分并重新组合了随机的 UUID，使用 `bit-or` 来合并当前的时间和 UUID 中最重要的 32 位。然后利用 `java.util.UUID` 的构造方法将这两部分 UUID 重新组合起来，得到了按时间顺序增长的 UUID：

```
(def first (squuid))
first
;; -> #uuid "527bf210-dfae-4c73-8b7a-302d3b511f41"

(def second (squuid))
second
;; -> #uuid "527bf219-65f0-4241-a165-c5c541cb98ea"

(def third (squuid))
third
;; -> #uuid "527bf232-42b2-44bc-8dd7-ddae2abfcb87"

(sort [first second third])
;; -> (#uuid "527bf210-dfae-4c73-8b7a-302d3b511f41"
;;     #uuid "527bf219-65f0-4241-a165-c5c541cb98ea"
;;     #uuid "527bf232-42b2-44bc-8dd7-ddae2abfcb87")
```

## 参阅

- 1.21 节“位操作”。
- 1.26 节“用字面值来表示日期”，探讨了 `#inst`，另一个读取程序的例子，用来表示日期。
- `java.util.UUID` 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/util/UUID.html）。

# 1.25 得到当前的日期和时间

作者：Ryan Neufeld

## 问题

需要得到当前的日期或时间。

## 解决方案

使用 Java 的 `java.util.Date`（http://docs.oracle.com/javase/7/docs/api/java/util/Date.html）构造方法，创建一个 `Date` 实例，代表当前的时间和日期：

```
(defn now []
  (java.util.Date.))

(now)
;; -> #inst "2013-04-06T14:33:45.740-00:00"

;; 几秒钟后……
(now)
;; -> #inst "2013-04-06T14:33:51.234-00:00"
```

如果对当前的 Unix 时间戳更感兴趣，请使用 System/currentTimeMillis：

```
(System/currentTimeMillis)
;; -> 1365260110635

(System/currentTimeMillis)
;; -> 1365260157013
```

## 讨论

对于 Clojure 来说，重新实现或包装 JVM 的时间和日期功能没有太大意义。因此，正常的做法是利用 Clojure 的 Java 互操作形式，实例化一个 Date 对象来表示“现在”。

#inst "2013-04-06T14:33:51.234-00:00" 看起来不太像 Java，对吗？这是因为 Clojure 的“时刻”读取程序字面值使用了 java.util.Date 作为其后台实现，在 1.26 节“用字面值来表示日期”中，我们探讨了更多关于 #inst 读取程序字面值的内容。

对于执行一次性的基准测试，使用 System/currentTimeMillis 可能很有用，但既然有一些高品质的工具来做这件事，currentTimeMillis 的作用就有限了。如果要做基准测试，也许可以试试 Hugo Duncan 的 Criterium 库（https://github.com/hugoduncan/criterium）。另外，不应该使用 currentTimeMillis 作为某种唯一的值，UUID 是更好的选择。

如果决定使用 clj-time（https://github.com/clj-time/clj-time）来处理日期，它提供 clj-time.core/now 来取得当前的 DateTime：

```
(require '[clj-time.core :as timec])

(timec/now)
;; -> #<DateTime 2013-04-06T14:35:15.453Z>
```

通过 clj-time.local/local-now 取得的 DateTime 实例，代表机器本地时区的当前时间：

```
(require '[clj-time.local :as timel])

(timel/local-now)
;; -> #<DateTime 2013-04-06T09:35:20.141-05:00>
```

## 参阅

- 1.24 节“生成唯一 ID”，探讨了如何生成通用唯一 ID。
- 1.26 节“用字面值来表示日期”，探讨了 #inst 读取程序常的更多信息。

# 1.26 用字面值来表示日期

作者：Ryan Neufeld

## 问题

需要将时间实例表示成可读的、可序列化的形式。

## 解决方案

使用 Clojure 的 `#inst` 字面值来表示某个时间点：

```
(def ryans-birthday #inst "1987-02-18T18:00:00.000-00:00")

(println ryans-birthday)
;; *out*
;; #inst "1987-02-18T18:00:00.000-00:00"
```

如果与其他 Clojure 进程通信（或任何使用 edn（https://github.com/edn-format/edn）的进程），请使用 `clojure.edn/read`，将时刻字面值字符串实例化为 `Date` 对象：

```
;; 一个假想的通信通道，" 接收 "edn 字符串
(require 'clojure.edn)
(import '[java.io PushbackReader StringReader])

(defn remote-server-receive-date []
  (-> "#inst \"1987-02-18T18:00:00.000-00:00\""
      (StringReader.)
      (PushbackReader.)))

(clojure.edn/read (remote-server-receive-date))
;; -> #inst "1987-02-18T18:00:00.000-00:00"
```

在前面的例子中，`remote-server-receive-date` 模拟了一个通信通道，通过它接收 edn 数据。

## 讨论

从 Clojure 1.4 开始，时刻是通过 `#inst` 读取程序字面值来表示的。这意味着日期不再用代码来表示并需要执行，而是有文本表示形式，既一致，又可序列化。对于所有能以 edn 进行通信的进程，这个标准允许它们清楚地说出时刻。支持 edn 的语言列表参见 edn 的实现列表（https://github.com/edn-format/edn/wiki/Implementations），目前包括 Clojure、Ruby 和 JavaScript，还有很多实现正在进行之中。

**`clojure.core/read` 与 `clojure.edn/read`**

虽然用 Clojure 内建的读取程序（`clojure.core/read`）似乎比较方便，但用这个读取程序从不信任的来源解析输入数据是不安全的。如果需要从外部来源读取简单的 Clojure 数据，最好是使用 edn 读取程序（`clojure.edn/read`）。

> clojure.core/read 不安全是因为其设计仅针对从信任的来源（比如你写的源文件）读取 Clojure 数据和字符串。clojure.edn/read 的设计特别针对使用通信通道的情况，设计时考虑了安全性。

也有可能改变数据读取程序，从而改变实例化 #inst 字面值的方式。如果需要，可以通过改变数据读取程序，将 #inst 字面值实例化为 java.util.Calendar（http://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html）或 java.sql.Timestamp（http://docs.oracle.com/javase/7/docs/api/java/sql/Timestamp.html）：

```
(def instant "#inst \"1987-02-18T18:00:00.000-00:00\"")

(binding [*data-readers* {'inst clojure.instant/read-instant-calendar}]
  (class (read-string instant)))
;; -> java.util.GregorianCalendar

(binding [*data-readers* {'inst clojure.instant/read-instant-timestamp}]
  (class (read-string instant)))
;; -> java.sql.Timestamp
```

### 参阅

- 1.24 节“ 生成唯一 ID”，探讨了 Clojure 中读取程序字面值的另一个例子。

## 1.27 利用clj-time解析日期和时间

作者：Ryan Neufeld

### 问题

需要从字符串中解析日期。

### 解决方案

直接使用 Java 的日期和时间类就像拔牙一样痛苦。我们建议使用 clj-time（https://github.com/clj-time/clj-time），它是优秀的 Joda-Time 库（http://joda-time.sourceforge.net/）的 Clojure 包装。

开始之前，请在项目依赖关系中加入 [clj-time "0.6.0"]，或用 lein-try 开始 REPL：

```
$ lein try clj-time
```

用 clj-time.format/formatter 来创建定制的日期 / 时间表示格式，能够解析待处理的字符串。调用 clj-time.format/parse，带上这些定制的格式，将字符串解析成 DateTime 对象：

```
(require '[clj-time.format :as tf])

;; 解析 "02/18/87" 这样的日期
(def government-forms-date (tf/formatter "MM/dd/yy"))

(tf/parse government-forms-date "02/18/87")
;; -> #<DateTime 1987-02-18T00:00:00.000Z>

(def wonky-format (tf/formatter "HH:mm:ss:SS' on 'yyyy-MM-dd"))
;; -> #'user/wonky-format

(tf/parse wonky-format "16:13:49:06 on 2013-04-06")
;; -> #<DateTime 2013-04-06T16:13:49.060Z>
```

## 讨论

`formatter` 函数是一个强大的小函数，接受日期/时间格式字符串，返回一个对象，该对象能解析那种格式的日期/时间字符串。这种格式字符串用符号来表示时间或日期的各个部分，而且能够包含任意数量的符号。符号的例子包括年（"yy" 或 "yyyy"）、日（"dd"），甚至是字面值字符串，如 "`on`"。这些符号的完整列表请参见 Joda-Time 的 `DateTimeFormat` 的 javadoc 文档（http://joda-time.sourceforge.net/apidocs/org/joda/time/format/DateTimeFormat.html）。

更常见的情况是，要解析的日期和时间有点奇怪，但又不是奇怪到从来没人见过。对此，`clj-time` 包含了大量的内建格式。调用 `clj-time.format/show-formatters` 可以打印出内建格式的列表，以及每种格式的示例日期/时间。如果选定了一种合适的格式，就可以调用 `clj-time.format/formatters`，带上它的关键字作为参数，得到相应的 `DateTimeFormatter`。

默认情况下，`formatter` 总是将字符串解析成 `DateTime` 对象，并带有 UTC 时区。`Formatter` 的第二个参数是可选的，表示时区。可以用 `clj-time.core/time-zone-for-offset` 或 `clj-time.core/time-zone-for-id` 得到一个 `DateTimeZone` 对象，传递给 `formatter`。

## 参阅

- 1.28 节“利用 `clj-time` 格式化日期”，探讨了如何利用格式类得到字符串。
- Java 简单日期格式类的官方 API 文档（http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html）。

# 1.28. 利用clj-time格式化日期

作者：Ryan Neufeld

## 问题

需要以特定格式打印日期或时间。

## 解决方案

虽然可以利用 clojure.core/format 来格式化 Java 日期相关的一些实例（Date、Calendar 和 Timestamp），但应该使用 clj-time（https://github.com/clj-time/clj-time）来格式化日期。

开始之前，请在项目依赖关系中加入 [clj-time "0.6.0"]，或用 lein-try 开始 REPL：

```
$ lein try clj-time
```

要将日期 / 时间输出为字符串，请调用 clj-time.format/unparse，并带上一个 DateTimeFormatter 参数。有一些内建的格式，可以通过 clj-time.format/formatters 获得，或者也可以利用 clj-time.format/formatter，建立自己的格式：

```
(require '[clj-time.format :as tf])
(require '[clj-time.core :as t])


(tf/unparse (tf/formatters :date) (t/now))
;; -> "2013-04-06"

(def my-format (tf/formatter "MMM d, yyyy 'at' hh:mm"))
(tf/unparse my-format (t/now))
;; -> "Apr 6, 2013 at 04:54"
```

## 讨论

当然有可能格式化纯的 Java 日期和时间，但根据我们的经验，不值得这么麻烦：语法又难看，步骤又繁琐。clj-time 及其背后的 Joda-Time 库有很好的表现记录，让 JVM 环境下的日期和时间处理变得容易。

formatter 函数是个好东西。它不仅能得到一个“格式”，用于打印或反解析日期，也能够将字符串解析成日期对象。换言之，DateTimeFormatter 能够在字符串和 Date 之间来回转换。1.27 节“利用 clj-time 解析日期和时间”介绍了 formatter 和 formatters 的许多工作原理。

有一种格式符号在解析时用得不多，它就是星期几的文本表示（如 "Tuesday" 或 "Tue"）。在格式字符串中用 "E" 来表示星期几的缩写，用 "EEEE" 来表示全称：

```
(def abbr-day (tf/formatter "E"))
(def full-day (tf/formatter "EEEE"))

(tf/unparse abbr-day (t/now))
;; -> "Mon"
(tf/unparse full-day (t/now))
;; -> "Monday"
```

如果需要格式化原生的 Java 日期 / 时间实例，可以使用 clj-time.coerce 命名空间中的函数，将 Java 日期 / 时间实例强制转换成 Joda-Time 实例：

```
(require '[clj-time.coerce :as tc])

(tc/from-date (java.util.Date.))
;; -> #<DateTime 2013-04-06T17:03:16.872Z>
```

类似地，也可以使用 `clj-time.coerce`，将 Joda-Time 实例强制转换成其他格式：

```
(tc/to-date (t/now))
;; -> #inst "2013-04-06T17:03:57.239-00:00"

(tc/to-long (t/now))
;; -> 1365267761585
```

## 参阅

- GitHub 上的 `clj-time` 项目主页（https://github.com/clj-time/clj-time）。
- 1.27 节"利用 `clj-time` 解析日期和时间"，探讨了 `formatter` 和 `formatters` 的更多细节。
- Java 简单日期格式类的官方 API 文档（http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html）。

# 1.29 比较日期

作者：Ryan Neufeld

## 问题

需要比较两个日期，或需要按日期排序。

## 解决方案

可以利用 `compare` 函数比较 Java 的 `Date` 对象：

```
(defn now [] (java.util.Date.))
(def one-second-ago (now))
(Thread/sleep 1000)

;; 现在比一秒前大（1）
(compare (now) one-second-ago)
;; -> 1

;; 一秒前比现在小（-1）
(compare one-second-ago (now))
;; -> -1

;; " 相等 " 用 0 表示
(compare one-second-ago one-second-ago)
;; -> 0
```

## 讨论

为什么不用 Clojure 内建的比较操作符（<=、> 等）来比较日期？这些操作符的问题在于，它们利用了 `clojure.lang.Numbers`，并试图将它们的参数强制转换成数值类型。

既然常规的比较不行，就有必要用 `compare` 函数。`compare` 函数接受两个参数，返回一个数值，表明第一个参数小于（–1）、等于（0）或大于（+1）第二个参数。

Clojure 的 `sort` 函数背后调用了 `compare`，所以对日期集合排序不需要额外的工作：

```
(def occurrences
  [#inst "2013-04-06T17:40:57.688-00:00"
   #inst "2002-12-25T00:40:57.688-00:00"
   #inst "2025-12-25T11:23:31.123-00:00"])

(sort occurrences)
;; -> (#inst "2002-12-25T00:40:57.688-00:00"
;;     #inst "2013-04-06T17:40:57.688-00:00"
;;     #inst "2025-12-25T11:23:31.123-00:00")
```

如果已经对日期和时间做过更复杂的工作，并且拥有 Joda-Time 对象，那么所有这些仍然有效。但是，如果需要比较 Joda-Time 对象和 Java 时间对象，就必须用 `clj-time.coerce` 中的函数，将它们强制转换成统一的类型。

## 参阅

- 2.24 节“值的比较与排序”。

# 1.30　计算时间间隔的长度

作者：Ryan Neufeld

## 问题

需要计算两个时间点之间的差值。

## 解决方案

由于 Java 的日期和时间类对时区和闰年的支持很弱，所以用 `clj-time` 库（https://github.com/clj-time/clj-time）来计算时间间隔的长度。

开始之前，请在项目依赖关系中加入 `[clj-time "0.6.0"]`，或用 `lein-try` 开始 REPL：

```
$ lein try clj-time
```

使用 `clj-time.core` 命名空间中的 `interval` 和众多的 `in-<unit>` 辅助函数，来计算两个时

间点之间的差值：

```
(require '[clj-time.core :as t])

;; 第一步是将两个日期作为一个 interval 记录下来
(def since-april-first
  (t/interval (t/date-time 2013 04 01) (t/now)))

;; dt 是 2013 年愚人节到今天之间的间隔
since-april-first
;; -> #<Interval 2013-04-01T00:00:00.000Z/2013-04-06T20:06:30.507Z>

(t/in-days since-april-first)
;; -> 5

;; 自登月以来的年数
(t/in-years (t/interval (t/date-time 1969 07 20) (t/now)))
;; -> 43

;; 2012 年 2 月 28 日到 3 月 1 日之间的天数（闰年）
(t/in-days (t/interval (t/date-time 2012 02 28)
                       (t/date-time 2012 03 01)))
;; -> 2

;; 平年的情况
(t/in-days (t/interval (t/date-time 2013 02 28)
                       (t/date-time 2013 03 01)))
;; -> 1
```

## 讨论

计算时间间隔的长度是比较复杂的时间操作。地球上的时间是一只复杂的怪兽，因为闰年和时区等结构而变得复杂。据我们所知，`clj-time`（https://github.com/clj-time/clj-time）是唯一能对付这种复杂性的库。

`clj-time.core/interval` 函数接受两个日期，返回离散的时间间隔的表示形式。而 `clj-time.core` 命名空间包含了大量的 `in-<unit>` 函数，可以用不同的单位来表示这段时间间隔。这些辅助函数的范围从 `in-msecs` 到 `in-years`，涵盖了一般应用程序中几乎所有有用的时间单位。

`clj-time` 不支持的一项功能是闰秒。Joda-Time 的官方 FAQ（http://joda-time.sourceforge.net/faq.html）解释了为什么没有这项功能。我们没有发现任何 Clojure 库能在这样的粒度上推演时间。如果这让你不爽，那么你可能是少数几个能把这事做对的人。祝你好运。

## 参阅

- 1.29 节“比较日期”。
- 1.31 节“生成一系列的日期和时间”。
- 1.33 节“根据日期间的关系取得日期”。

# 1.31 生成一系列的日期和时间

作者：Ryan Neufeld

## 问题

需要生成一个惰性序列（lazy sequence），包含一系列的日期和（或）时间。

## 解决方案

这个问题在 Java 中没有容易的解决方案，在 Clojure 中也没有，包括第三方的库。但是，可以通过 `clj-time`（https://github.com/clj-time/clj-time）来近似。通过组合使用 `clj-time` 的 `Interval` 和 `periodic-seq` 功能，可以创建一个函数 `time-range`，模拟 `range` 的功能，但针对的是 `DateTime`：

```
(require '[clj-time.core :as time])
(require '[clj-time.periodic :as time-period])

(defn time-range
  "Return a lazy sequence of DateTimes from start to end, incremented
  by 'step' units of time."
  [start end step]
  (let [inf-range (time-period/periodic-seq start step)
        below-end? (fn [t] (time/within? (time/interval start end)
                                         t))]
    (take-while below-end? inf-range)))
```

下面是 `time-range` 函数的用法：

```
(def months-of-the-year (time-range (time/date-time 2012 01)
                                    (time/date-time 2013 01)
                                    (time/months 1)))

;; months-of-the-year 是未实现的惰性序列
(realized? months-of-the-year)
;; -> false

(count months-of-the-year)
;; -> 12

;; 现在实现了
(realized? months-of-the-year)
;; -> true
```

## 讨论

在 Clojure 中，尽管没有准备好的、立即可用的 `time-range` 解决方案，但创建这样一个具备纯惰性语义的函数并不困难。我们的惰性 `time-range` 函数的基础是一个无限的值序列，

从一个固定的时间开始：

```
(defn time-range
  "Return a lazy sequence of DateTimes from start to end, incremented
  by 'step' units of time."
  [start end step]
  (let [inf-range (time-period/periodic-seq start step)           ; ❶
        below-end? (fn [t] (time/within? (time/interval start end) ; ❷
                                         t))]
    (take-while below-end? inf-range)))                            ; ❸
```

❶ 取得一个惰性无限序列。

❷ 创建一个谓词来终止该序列。

❸ 修改该序列，当 `below-end?` 失败时终止（当然也是惰性的）。

调用 `periodic-seq`，带上 `start` 和 `step`，就会返回一个无限的惰性值序列，从 `start` 开始，下一个值比上一个值增长一个 `step`。

拥有惰性序列是一回事，但我们还需要一种惰性的方式，在到达 `end` 时停止取值。在 `let` 中创建的 `below-end?` 函数利用 `clj-time.core/interval` 来构建从 `start` 到 `end` 的时间间隔，并用 `clj-time.core/within?` 来测试时间 `t` 是否在这个间隔之内。这个函数被当作谓词传递给 `take-while`，它将惰性地消费这些值，直到 `below-end?` 失败。

总之，`time-range` 返回一个惰性的 `DateTime` 对象序列，从一个开始时间到一个结束时间，步长是提供的 `step` 值。

想象一下，用一种不具备一级惰性的语言来构建类似的东西会是什么结果。

## 参阅

- 1.29 节“比较日期”。
- 1.30 节“计算时间间隔的长度”。
- 1.32 节“利用原生 Java 类型生成一系列日期和时间”，探讨了另一种只使用原生类型的方式。
- 1.33 节“根据日期间的关系取得日期”。

# 1.32 利用原生Java类型生成一系列日期和时间

作者：Tom Hicks

## 问题

需要生成一个惰性序列，包含一系列的日期和时间。而且，不像 1.31 节那样，而是希望只用内建的类型。

## 解决方案

可以使用 Java 的 `java.util.GregorianCalendar` 类（http://docs.oracle.com/javase/7/docs/api/java/util/GregorianCalendar.html），加上 Clojure 的 `repeatedly` 函数，生成格里高利日历日期的惰性序列。然后用 `java.text.SimpleDateFormat`（http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html）来格式化这些日期，有大量输出格式可选。

这个例子生成了格里高利[4]日历日期的无限惰性序列，从 1970 年 1 月 1 日开始，一天接一天。然后用核心的 `take` 和 `drop` 函数，选择 2 月的最后两天（注意，不要在 REPL 中对无限序列估值）：

```
(def daily-from-epoch
  (let [start-date (java.util.GregorianCalendar. 1970 0 0 0 0) ]
    (repeatedly
      (fn []
        (.add start-date java.util.Calendar/DAY_OF_YEAR 1)
        (.clone start-date)))))

(take 2 (drop 57 daily-from-epoch))
;; -> (#inst "1970-02-27T00:00:00.000-07:00"
;;     #inst "1970-02-28T00:00:00.000-07:00")
```

## 讨论

Clojure 没有自己的日期类型，默认情况下，它靠的是容易与 Java 互操作的能力（但请参考 `clj-time` 库，它提供了 Java 日期、时间和日历类的替代）。

这个解决方案基于核心的 `repeatedly` 函数，它通过重复调用传给它的参数函数，得到一个惰性序列，就是函数的返回结果构成的序列。因为没有向 `repeatedly` 提供可选的、有限的参数，所以结果序列是无限的。因此，在 REPL 环境中，必须谨慎地在上下文中（例如 `take` 和 `drop`）评估结果序列，限制生成的值。

因为向 `repeatedly` 提供的函数是没有参数的，所以假定是通过副作用来实现它的目标（这使它成为非纯函数）。这里，当参数函数创建一个格里高利日历，并反复地让它以一天为单位递增时，非纯就发生了。每次调用该函数，它返回那个格里高利日历对象的拷贝（为了避免神秘的、不期望的副作用，建议不要直接返回那个变化的对象）。

结果序列中的日期值是 `java.util.GregorianCalendar` 类型，但 REPL 的 `print` 函数将它们显示为 `#inst` 读取程序字面值。可以将 `class`（或 `type`）函数映射到这个序列上，验证该序列的元素是格里高利日历对象：

---

注 4："格里高利"是我们都知道而且喜欢的日历的正式名称。更多内容请参考维基百科（http://en.wikipedia.org/wiki/Gregorian_calendar）。

```
(def end-of-feb (take 2 (drop 57 daily-from-epoch)))
(map class end-of-feb)
;; -> (java.util.GregorianCalendar java.util.GregorianCalendar)
```

可以将这个解决方案一般化，变成一个接受一个起始年份参数的函数，但如果没提供该参数，就默认为某个方便的年份：

```
(defn daily-from-year [& [start-year]]
  (let [start-date (java.util.GregorianCalendar. (or start-year 1970)
                                                 0 0 0 0)]
    (repeatedly
      (fn []
        (.add start-date java.util.Calendar/DAY_OF_YEAR 1)
        (.clone start-date) ))))

(take 3 (daily-from-year 1999))
;; -> (#inst "1999-01-01T00:00:00.000-07:00"
;;     #inst "1999-01-02T00:00:00.000-07:00"
;;     #inst "1999-01-03T00:00:00.000-07:00")

(take 2 (daily-from-year))
;; -> (#inst "1970-01-01T00:00:00.000-07:00"
;;     #inst "1970-01-02T00:00:00.000-07:00")
```

利用 java.text.SimpleDateFormat 类，可以将日期格式化为各种不同的格式：

```
(def end-of-days (take 3 (drop 353 (daily-from-year 2012))))
(def cal-format (java.text.SimpleDateFormat. "EEE M/d/yyyy"))
(def iso8601-format (java.text.SimpleDateFormat. "yyyy-MM-dd'T'HH:mm:ss'Z'"))

(map #(.format cal-format (.getTime %)) end-of-days)
;; -> ("Wed 12/19/2012" "Thu 12/20/2012" "Fri 12/21/2012")

(map #(.format iso8601-format (.getTime %)) end-of-days)
;; -> ("2012-12-19T00:00:00Z" "2012-12-20T00:00:00Z" "2012-12-21T00:00:00Z")
```

综合起来，创建一个函数来生成无限惰性序列，包含格式化的格里高利日期字符串。为了方便，该函数接受可选的起始年份和日期格式字符串参数：

```
(defn gregorian-day-seq
  "Return an infinite sequence of formatted Gregorian day strings
  starting on January 1st of the given year (default 1970)"
  [& [start-year date-format]]
  (let [gd-format (java.text.SimpleDateFormat. (or date-format "EEE M/d/yyyy"))
        start-date (java.util.GregorianCalendar. (or start-year 1970) 0 0 0 0)]
    (repeatedly
      (fn []
        (.add start-date java.util.Calendar/DAY_OF_YEAR 1)
        (.format gd-format (.getTime start-date)) ))))
```

要测试该函数，找到一年中所有的星期日，再选出最后一个。

```
(def y2k (take 366 (gregorian-day-seq 2000)))
(last (filter #(.startsWith % "Sun") y2k))
;; -> "Sun 12/31/2000"
```

## 参阅

- 1.25 节“得到当前的日期和时间”，探讨了在 Clojure 中使用 `java.util.Date`。
- 1.26 节“用字面值来表示日期”，介绍了 Clojure 中针对日期 / 时间的 `#inst` 读取程序字面值。
- 1.31 节“生成一系列的日期和时间”，介绍了另一种使用 `clj-time`/Joda-Time 的方法。

# 1.33 根据日期间的关系取得日期

作者：Ryan Neufeld

## 问题

需要根据某个时间来算出另一个时间，就像 Ruby on Rails（http://rubyonrails.org/）中的 `2.days.from_now`。

## 解决方案

因为相对时间是一只复杂的怪兽，所以我们建议用 `clj-time`（https://github.com/clj-time/clj-time）来计算相对日期和时间。

开始之前，请在项目依赖关系中加入 `[clj-time "0.6.0"]`，或用 `lein-try` 开始 REPL：

```
$ lein try clj-time
```

如果你用过 Ruby on Rails 框架，那么可能熟悉这样的语句：`1.day.from_now`，`3.days.ago` 或 `some_date - 2.years`。你会很高兴 `clj-time` 提供了类似的功能：

```
(require '[clj-time.core :as t])

;; 1.day.from_now（写这个程序是在 4 月 6 号）
(-> 1
    t/days
    t/from-now)
;; -> #<DateTime 2013-04-07T20:36:52.012Z>

;; 3.days.ago
(-> 3
    t/days
    t/ago)
;; -> #<DateTime 2013-04-03T20:37:06.844Z>
```

clj-time.core 的函数 from-now 和 ago 只是语法糖，背后是 plus 和 minus：

```
;; 1.day.from_now
(t/plus (t/now) (t/years 1))
;; -> #<DateTime 2014-04-06T20:41:43.638Z>

;; some_date - 2.years
(def some-date (t/date-time 2053 12 25))
(t/minus some-date (t/years 2))
;; -> #<DateTime 2051-12-25T00:00:00.000Z>
```

## 讨论

尽管日期和时间处理有时候在 Java 中很难，但 clj-time 设法提供了容易的语法，实现日期的加减。

函数 plus、minus、from-now 和 ago 都接受一段时间间隔，并根据它来调整一个 DateTime（这个时间可以是“现在”，就像在 from-now 或 ago 中那样，也可以是某个给定的时间点）。

clj-time.core 包含了一些有用的时间间隔辅助类，从 millis 到 years，以给定的尺度给出时间间隔。

根据使用情况，甚至可以将操作、时间间隔和时间巧妙地加以安排，读起来就像一句话。

以 (-> 1 t/years t/from-now) 为例。在这个例子中，线索宏 -> 将值串起来，每个作为后一个的参数，得到 (t/from-now (t/years 1))。

可以根据自己的喜好，安排函数调用，但要知道，完全能够像这样产生可读的、深度嵌套的调用。

## 参阅

- 1.29 节“比较日期”。
- 1.31 节“生成一系列的日期和时间”。

# 1.34　处理时区

作者：Ryan Neufeld

## 问题

需要优雅地处理一些时区中的时间和日期。

## 解决方案

JVM 内建的时间和日期类与时区的表示法配合不好。比如说，Date 将每个值都当成 UTC，而 Calendar 在 Clojure 中用起来麻烦（或者说在 Java 中，就这件事来说）。请用 clj-time（https://github.com/clj-time/clj-time）来适当地处理时区。

开始之前，请在项目依赖关系中加入 [clj-time "0.6.0"]，或用 lein-try 开始 REPL：

```
$ lein try clj-time

(require '[clj-time.core :as t])

;; 我的生日，在正确的时区
(def bday (t/from-time-zone (t/date-time 2012 02 18 18)
                            (t/time-zone-for-offset -6)))

bday
;; -> #<DateTime 2012-02-18T18:00:00.000-06:00>

;; 我出生时布里斯班是什么时间？
(def australia-bday
  (t/to-time-zone bday (t/time-zone-for-id "Australia/Brisbane")))

australia-bday
;; -> #<DateTime 2012-02-19T10:00:00.000+10:00>

;; 但它们是同一时刻
(compare bday australia-bday)
;; -> 0
```

## 讨论

不像 Java 内建的类，clj-time 知道时区。clj-time 包装的库 Joda-Time（http://joda-time.sourceforge.net/），包含了国际公认的 tz 数据库（http://www.twinsun.com/tz/tz-link.htm）。这个数据库记录了地球上几乎每个地点的 ID 和时间偏移。

tz 数据库也记录了夏令时的信息。例如，洛杉矶在冬天是 UTC-08:00，在夏天是 UTC-07:00。这在使用 clj-time 时得到了准确的反映：

```
(def la-tz (t/time-zone-for-id "America/Los_Angeles"))

;; LA 在冬天是 UTC-08:00
(t/from-time-zone (t/date-time 2012 01 01) la-tz)
;; -> #<DateTime 2012-01-01T00:00:00.000-08:00>

;; ... 在夏天是 UTC-07:00
(t/from-time-zone (t/date-time 2012 06 01) la-tz)
;; -> #<DateTime 2012-06-01T00:00:00.000-07:00>
```

clj-time.core/from-time-zone 函数接受任何 DateTime，并将它的时区改为期望的时区。

如果分别收到日期、时间和时区，希望将它们合成准确的 DateTime，就可以用它。

clj-time.core/to-time-zone 函数和 from-time-zone 的参数列表一样，它返回的 DateTime 就是同一个时间点，但是采用另一个时区的视角。要将不同来源的时间和日期信息提供给客户，并按照客户喜欢的时区，就可以用它。

有时候你可能只想处理机器本地时间。clj-time.local 命名空间提供了一些函数，包括 local-now，用来取得本地时区的时间，也包括 to-local-date-time，将时间的视角转换成本地时区。

## 参阅

- 1.30 节“计算时间间隔的长度”，以及 1.33 节“根据日期间的关系取得日期”。

# 1.35　将Unix时间戳转换成Date对象

作者：Steven Proctor

## 问题

邮

电

需要从 Unix 时间戳得到一个 Date 对象。

## 解决方案

在处理来自外部系统的数据时，你会发现许多系统以 Unix 时间格式来表示时间戳。在处理某些数据库时，从日志文件的时间戳中解析数据时，或与其他一些系统打交道，它们需要处理跨不同时区和文化的日期与时间，你可能会遇到这种情况。

幸运的是，利用 Clojure 与 Java 的良好互操作能力，有容易的解决方案：

```
(defn from-unix-time
  "Return a Java Date object from a Unix time representation expressed
  in whole seconds."
  [unix-time]
  (java.util.Date. unix-time))
```

下面是如何使用 from-unix-time 函数：

```
(from-unix-time 1366127520000)
;; -> #inst "2013-04-16T15:52:00.000-00:00"
```

## 讨论

要从 Unix 时间对象得到一个 Java 的 Date 对象，只要利用 Clojure 的 Java 互操作功能，

构造一个新的 java.util.Date 对象（http://docs.oracle.com/javase/7/docs/api/java/util/Date.html）。

如果已经使用 clj-time 库（https://github.com/clj-time/clj-time）或想用它，就可以用 clj-time，从 Unix 时间戳得到一个 DateTime 对象：

```
(require '[clj-time.coerce :as timec])

(defn datetime-from-unix-time
  "Return a DateTime object from a Unix time representation expressed
  in whole seconds."
  [unix-time]
  (timec/from-long unix-time))
```

应用 datetime-from-unix-time 函数，就会发现得到了一个 DateTime，包含了正确的时间：

```
(datetime-from-unix-time 1366127520000)
;; -> #<DateTime 2013-04-16T15:52:00.000Z>
```

平时也许不需要担心日期和时间表示成秒的形式，但如果需要，知道时间戳很容易转成日期格式，用于系统的其他部分，不是很好吗？

## 参阅

- 1.25 节“得到当前的日期和时间”。
- 1.36 节“将 Date 对象转换成 Unix 时间戳”。

# 1.36　将Date对象转换成Unix时间戳

作者：Steven Proctor

## 问题

需要从一个 Date 对象得到 Unix 时间戳的表示形式。

## 解决方案

许多系统以 Unix 时间的格式来表示时间戳，如果你必须与这些系统打交道，就必须以它们期望的格式给出日期和时间信息。

幸运的是，利用 Clojure 与 Java 的良好互操作能力，有容易的解决方案：

```
(defn to-unix-time
  "Returns a Unix time representation expressed in whole seconds
   given a java.util.Date."
```

```
  [date]
  (.getTime date))
```

下面是如何使用 to-unix-time 函数：

```
(def date (read-string "#inst \"2013-04-16T15:52:00.000-00:00\""))
;; -> #'user/date

(to-unix-time date)
;; -> 1366127520000
```

## 讨论

如果有一个 java.util.Date（http://docs.oracle.com/javase/7/docs/api/java/util/Date.html）对象，那么利用 Clojure 提供的 Java 互操作性，很容易将时间表示为 Unix 时间。Java 的 Date 对象有一个方法，名为 getTime，返回 Unix 格式的时间。

如果已用或想用 clj-time 库（https://github.com/clj-time/clj-time），就可以利用 clj-time 从 DateTime 对象得到 Unix 时间格式的时间：

```
(require '[clj-time.coerce :as timec])

(defn datetime-to-unix-time
  "Returns a Unix time representation expressed in whole seconds
   given a DateTime."
  [datetime]
  (timec/to-long datetime))
```

应用 datetime-to-unix-time 函数，会发现从 DateTime 对象得到了一个 Unix 时间：

```
(def datetime (clj-time.core/date-time 2013 04 16 15 52))
;; #'user/datetime

(datetime-to-unix-time datetime)
;; 1366127520000
```

有了 clj-time.coerce，只要使用 to-long 函数就可以将 Joda-Time 的 DateTime 对象转换成 Unix 时间格式。

或许你的系统永远不会与那些需要 Unix 时间戳的系统打交道，但如果设计的系统有这种需要，Clojure 能够容易地将 Date 或 DateTime 表示成 Unix 时间格式。

## 参阅

- 1.25 节“得到当前的日期和时间”。
- 1.35 节“将 Unix 时间戳转换成 Date 对象”。

# 第2章
# 复合数据

## 2.0 简介

既然我们已经掌握了原生类型，就可以用它们做点事情了。单个原子值很好，但如果将它们聚在一起，事情就会变得更加有趣。你很快会看到，数据操作是 Clojure 的一个强项。

是什么让 Clojure 如此擅长操作集合？原因有三点：*不变性、持久性和序列抽象*。Clojure 内建的每个集合类型都有这些特性，因此在 API 外观和行为上是一致的。

正如伟大的 Alan J. Perlis（早期计算机科学先驱）所说的：

> *一种数据结构上有 100 个函数操作，胜过 10 种数据结构上有 10 个函数操作。*

本章介绍了 Clojure 的集合，以及在什么情况下如何使用它们。最后，我们利用 Clojure 的接口多态能力，通过构建一个外观和行为与其他 Clojure 集合类似的特征完备的类型，来总结这一章的内容。

### 不变性

不变性意味着 Clojure 的数据结构一旦创建，就永远不能改变。要“修改”一个不能改变的数据结构，只能复制老数据结构，并在相应位置作出期望的修改，从而创建一个新的数据结构。

不变性也意味着，Clojure 的数据结构不论嵌套多深，都是简单的值，就像数字 `3` 或字符 `\z`。

说“改变”`3`的值是没什么意义的，因为它就是`3`。如果“改变”是指增加它，也不会去改变`3`本身。相反，会得到完全不同的新值，即`4`。Clojure 将这个值的概念推广到所有的数据结构。对 Clojure 来说，其他语言中更新数据结构的操作，都会返回一个全新的数据结构。你可以继续充满自信地同时传递并使用老的版本和新的版本，因为你知道自己所做的任何事情都不会导致程序在别的地方发生意想不到的变化。

这个特性在并发和并行编程中非常重要，因为意外的变化是一大类缺陷的来源。有了不变的数据，不论多少线程都可以读取同一数据，不用担心锁或竞争条件，因为读取不能改变的东西总是安全的。“复制”操作不仅没有成本，也没有必要。

## 持久性

你可能会问，“不变性”会有效率吗？每当要加点东西的时候都要复制对象，这一定不切实际，不是吗？

是的，也许会这样，但持久性避免了这一点。持久性意味着 Clojure 的数据结构尽管在逻辑上是不变的，但仍能共享其内部结构的一些部分，以实现时间和空间上的效率。基本上，更新不变数据的版本只需要更新相对以前版本的变化，而不是完整的深复制。

当然，为了保证性能，这种机制使用了一些极其巧妙的算法。它们的工作原理请参考 Chris Okasaki 的著作 *Purely Functional Data Structures*（Cambridge University Press）。

## 序列抽象

从向量、映射、集、列表到字符串和流，每个 Clojure 集合的行为都是类似的、可预测的。它们都是划时代的简单高效的工具。这靠的是 Clojure 的序列抽象。

Clojure 中的集合操作函数家族，其实现都基于一个简单抽象：每个集合都可以看成值的序列。通过实现`first`、`rest`和`cons`，所有数据结构（甚至你自己建立的）都可以分享`ISeq`接口。

然后，就可以使用 Clojure 中大量的序列操作函数以及所有的数据结构了。所有函数式编程最让人喜爱的函数（`map`、`reduce`、`filter`等），都可以互换应用在所有数据结构上。从本质上说，序列抽象既提供了传统基于列表的 LISP 编程的全部表达能力，又不强迫你真正去用列表。相反，你可以用对任务最有效的类型，而且只要你觉得有必要，就可以用同样的方式使用它们。

# 2.1 创建列表

作者：Luke VanderHart

## 问题

需要在源代码中创建一个列表数据结构。

## 解决方案

要明确地创建一个列表（`clojure.lang.PersistentList`），有两种基本方法。

可以用单引号加上括号，表明这个列表应该被当成一个数据结构，不要马上求值：

```
'(1 :2 "3")
;; -> (1 :2 "3")
```

或者，更常见的是，可以使用 `list` 函数，它接受可变数目的参数，用它们创建一个列表：

```
(list 1 :2 "3")
;; -> (1 :2 "3")
```

## 讨论

通常，在这两种方法中，使用 `list` 函数是更好的选择。用单引号列表的问题在于，它阻止了列表中所有表达式的求值，这意味着这些符号会被当作字面符号返回，而不是解析变量或调用函数。但 `list` 会以正常的方式对参数求值，然后再创建列表。对于不是宏的代码，这通常是期望的方式：

```
(def x 2)

'(1 x)
;; -> (1 x)

(list 1 x)
;; -> (1 2)
```

这就是说，`'()` 是创建空列表的习惯方式，它更简明。由于它是空的，也不用考虑内容求值。

### 列表与向量

Clojure 既有列表（list），也有向量（vector）。两者都是序列数据结构。但对大多数情况，向量更适合，在 Clojure 的习惯中更常用。

这有一些原因。向量的字面语法比列表更清楚，空间效率和性能是一样的。而且，向量利用索引，支持接近常数的查找时间 ($O(\log_{32} n)$)，而列表则不同，需要线性地查找时间 ($O(n)$)。

一般来说，明确地选择列表而非向量只有一个原因，就是需要数据结构支持高效的前端插入，列表能做到这一点，而向量在尾端添加元素是最高效的。

## 参阅

- 2.2 节“从已有的数据结构创建列表”。
- 2.6 节“创建向量”。

# 2.2 从已有的数据结构创建列表

作者：Luke VanderHart

## 问题

已有一个序列数据结构，需要将它转换成列表，作为具体的数据类型。

## 解决方案

最容易的解决方案是：不要这么做。拥有具体的列表并不能提供多少好处，不如直接使用已有数据结构的序列抽象，而且对于大型数据结构，转换代价可能很高。

如果确实需要显式地转换具体的数据结构，那么有两种方法。

首先，可以使用 `apply` 函数来调用 `list` 函数，将已有的数据结构作为它的参数：

```
(apply list [1 2 3 4 5])
;; -> (1 2 3 4 5)
```

或者，可以使用 `into` 函数，反复连接取自原有数据结构的元素，得到列表。但要注意，这种方法的副作用是颠倒原来集合的顺序：

```
(into '() [1 2 3 4 5])
;; -> (5 4 3 2 1)
```

## 讨论

这两种方法都可行。但其工作原理差异非常大。

在使用 `apply` 时，实际上是调用 `list` 函数，而它的许多参数是放在原来的数据结构中。这听起来可能有点奇怪，如果原来的数据结构包含上百万个元素，就更奇怪了。用上百万个参数调用一个函数是意味着什么？鉴于 JVM 限制了方法不能超过 255 个参数（参见 JVM 类文件规格说明（http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.3.3）），这怎么可行呢？

实际上，带有可变参数的函数（例如 `list`），其处理方式有些特殊：参数列表是作为一个序列传入的。`apply` 知道这一点，将原来数据结构的序列视图直接传给了接受参数的函数。

这就是可行的原因，实际上没有一个 JVM 方法调用了上百万个参数。

into 的工作原理完全不同：它接受两个参数，第一个是一个数据结构，第二个是一个序列。它利用 conj 函数（将在别处详细讨论），反复地将序列中的元素结合到数据结构中。这就是序列颠倒的原因。元素是从序列的前端取出的，但列表的 conj 操作将它放在已有元素的前面。因此，输入序列的第一个元素将成为列表的最后一个元素，以此类推。

既然次序会颠倒，为什么要选择 into，而不是 apply ？答案是“速度”。into 利用了 Clojure 的瞬态数据结构（transient），它大幅提升了性能。在作者的机器上，利用 apply 将百万元素的向量转换成列表，平均耗时 750 毫秒，而使用 into 只花了不到一半的时间，平均耗时 350 毫秒。当然，列表的次序反过来了，不论将输入的次序还是输出的次序反过来，都会抵消速度的优势。最后，只有在颠倒的次序可以接受时，into 才有优势。

## 参阅

- 2.1 节“创建列表”。

# 2.3 在列表中“添加”一个元素

作者：Luke VanderHart

## 问题

需要在列表中添加一个元素，或者用函数式的术语来说，需要从已有的列表导出一个新列表，包含一个添加的元素。

## 解决方案

使用 conj 函数。conj 用于向逻辑集合添加一个或多个元素，它是多态的，这意味着它适用于多个具体的数据类型，包括列表：

```
(conj (list 1 2 3) 4)
;; -> (4 1 2 3)
```

也可以一次添加多个元素：

```
(conj (list 1 2 3) 4 5)
;; -> (5 4 1 2 3)
```

## 讨论

conj 的行为可能因具体的类型而稍有不同。它总是向不可变的集合“添加”一个元素，然

后返回一个包含那个元素的新集合。但它可能将新元素添加在集合的不同位置，这取决于在具体的类型中怎样做最高效。

如果是列表，conj 将在列表头上添加元素，因为链接列表数据结构仅在头上支持常量时间的插入。

> **conj 与 cons**
>
> 熟悉 Common Lisp 或 Scheme 的人，可能希望看到用 cons 函数，而不是 conj。Clojure 确实有 cons 函数，但它的目的有一点不同。
>
> conj 将返回一个新的具体类对象 clojure.lang.PersistentList（在用于列表时），而 cons 总是返回一个新的序列，元素加在第一个位置，后面是原来的集合。
>
> 区别很细微，尤其是从算法上来看，clojure.lang.PersistentList 和 cons 构造的序列都是持久的链接列表类型。
>
> 简而言之，conj 是具体的数据结构操作，它不会改变处理的数据结构的具体类型，而 cons 是序列操作，只保证它将返回一个序列：实际上，它返回一个 cons 单元（clojure.lang.Cons），实现了序列接口，不论给它的是什么类型的序列集合。
>
> 与 conj 不同，cons 也保证在返回的序列中，新增的元素放在头上，不论集合的类型是什么。

## 参阅

- 2.7 节“在向量中‘添加’一个元素”。

# 2.4　从列表中“移除”一个元素

作者：Luke VanderHart

## 问题

需要从列表中移除一个元素，得到不包含该元素的列表。

## 解决方案

从列表中删除第一个元素很容易，可以使用函数 rest 或 pop。在用于非空列表时，它们的效果相同：

```
(pop '(1 2 3))
;; -> (2 3)
```

```
(rest '(1 2 3))
;; -> (2 3)
```

## 讨论

rest 实际上是一个序列函数，用于得到序列的尾。由于 Clojure 的列表直接实现了序列接口，对列表使用 rest 总会返回另一个列表（可能是空的）。

pop 与 conj 类似，它操作具体的数据结构，而不是序列接口。像 conj 一样，它是多态的，移除元素的位置取决于什么对于这种具体类型最有效。

在用于空列表时，这两个函数的反应是不同的。pop 会抛出异常，而 rest 会返回空列表：

```
(pop '())
;; -> IllegalStateException Can't pop empty list ...

(rest '())
;; -> ()
```

除了第一个位置，列表不支持从其他位置移除元素。如果需要从列表中间或末尾移除一个元素，就需要使用序列操作函数，然后将结果转回具体的列表（如果因为某些原因绝对需要一个列表）。

## 参阅

- 2.8 节“从向量中‘移除’一个元素”。

# 2.5 测试是否列表

作者：Steve Miner

## 问题

需要测试一个值是不是一个列表。

## 解决方案

list? 函数似乎是明显的选择，但在大多数情况下，最好是使用更常用的 seq? 函数来测试。

## 讨论

list? 专门测试参数是否实现了 clojure.lang.IPersistentList 接口，但在大多数情况下，

实际上是想知道这个值是否是一个序列（实现 clojure.lang.ISeq 接口），它是比列表更常见的抽象。

并非所有打印出来的像列表（在括号中）的东西都能通过 list? 测试。在实践中，常常在操作列表时得到 Cons 和 LazySeq 值。你只要专注于基本的序列抽象，就不必担心这些具体实现的细节：

```
;; 通过 list 构建的列表同时通过 list? 和 seq? 测试
(list? (list 1 2 3))
;; -> true
(seq? (list 1 2 3))
;; -> true

;; cons 虽然看起来像列表，但实际上是 Cons 实例
(list? (cons 1 '(2 3)))
;; -> false
(type (cons 1 '(2 3)))
;; -> clojure.lang.Cons
(seq? (cons 1 '(2 3)))
;; -> true

;; range 的惰性返回值是序列，但不是列表
(list? (range 3))
;; -> false
(seq? (range 3))
;; -> true
(type (range 3))
;; -> clojure.lang.LazySeq
```

用 seq? 几乎总是比用 list? 好。

## 参阅


- 2.1 节“创建列表”。
- 2.3 节“在列表中‘添加’一个元素”。
- 2.26 节“检测集合是否包含几个值中的一个”。


# 2.6 创建向量

作者：Luke VanderHart

## 问题

需要创建一个向量数据结构，要么作为字面值，要么来自原有的数据结构。

## 解决方案

到目前为止，创建向量最容易的方式，就是使用字面值向量的方括号表示法。但是，也可以使用 `vector` 函数，它将参数创建为一个向量：

```
[1 :2 "3"]
;; -> [1 :2 "3"]

(vector 1 :2 "3")
;; -> [1 :2 "3"]
```

要通过已有的数据结构来构建，可以使用 `vec` 函数，它接受任何集合，返回包含同样元素的向量：

```
(vec '(1 :2 "3"))
;; -> [1 :2 "3"]
```

或者，可以使用 `into` 函数，它接受两个集合，用来自第二个集合的元素，对第一个集合反复调用 `conj`：

```
(into [] '(1 :2 "3"))
;; -> [1 :2 "3"]
```

## 讨论

很少有使用 `vector` 函数而不是字面值向量的情况。在 Clojure 中，向量不像列表，不会作为函数调用（或任何其他东西）来求值，所以不像列表字面值那样要考虑引号的问题。

奇怪的是，在利用已有的集合构建向量时，使用 `into` 要比使用 `vec` 快 30%，原因是用了瞬态数据结构来提速。如果需要转换大的集合，并且在意速度，就考虑使用 `into`；否则，`vec` 通常可读性更好。

## 参阅

- 2.1 节“创建列表”。
- 2.2 节“从已有的数据结构创建列表”。

# 2.7　在向量中“添加”一个元素

作者：Luke VanderHart

## 问题

需要在向量中添加一个元素，得到包含该元素的新向量。

## 解决方案

conj 函数在用于向量时，返回一个向量，末尾附加一个或多个元素：

```
(conj [1 2 3] 4)
;; -> [1 2 3 4]

(conj [1 2 3] 4 5)
;; -> [1 2 3 4 5]
```

## 讨论

除了末尾，向量不支持在其他任何地方添加新的元素。如果需要在中间插入一个元素，就需要使用序列操作函数，并在完成后将它转回成向量（如果有必要）。

由于向量是关联的（整数索引映射到值），也可以使用 assoc 函数，令索引等于向量当前的长度（最大索引加 1），来添加元素：

```
(assoc [:a :b :c] 3 :x)
;; -> [:a :b :c :x]
```

但是，这种方法比 conj 更容易出错。如果提供的索引太小，就可能“覆写”向量中前面的值，如果索引大于向量当前的长度，就会抛出 IndexOutOfBoundsException。

不过，这个方法仍有用武之地。如果代码已经对向量使用了 assoc，可以用这个技巧来更新值，得到新的向量。

## 参阅

- 2.3 节“在列表中‘添加’一个元素”。
- 2.6 节“创建向量”。

# 2.8　从向量中“移除”一个元素

作者：Luke VanderHart

## 问题

需要从向量中移除一个元素，得到不包含该元素的新向量。

## 解决方案

要有效地从向量末尾移除元素，请使用 pop 函数，它接受一个向量，返回没有末尾元素的新向量。

```
(pop [1 2 3 4])
;; -> [1 2 3]
```

## 讨论

尽管没有像 pop 移除末尾元素那样，专门从向量头上移除元素的操作，但有一个函数 subvec，可以有效地移除向量头部或尾部的任意个元素。给定一个向量，一个起始索引，一个（可选的）终止索引，它将返回从起始（包含）到终止（不包含）索引的向量。

下面的例子丢弃了向量头上的一个元素。可以像这样用 subvec：

```
(subvec [:a :b :c :d] 1)
;; -> [:b :c :d]
```

或者，移除向量头上和末尾的元素，将终止索引传递给 subvec：

```
(subvec [:a :b :c :d] 1 3)
;; -> [:b :c]
```

因为 subvec 利用了向量的内部表示，创建的子向量与原来的内部结构一样，它非常有效，执行时间是常数。它是从向量头上移除元素的唯一有效方法。

虽然可以在向量上使用 rest 或 drop 等函数，但它们在技术上是序列操作，不是向量操作。它们返回的值只能保证是序列，而不是具体的向量，因此也不能保证其具有向量的特点或性能。

当然，可以利用 vec 或 into []，将序列转换回具体的向量，但这对于大向量是代价高昂的操作。

## 参阅

- 2.4 节“从列表中‘移除’一个元素”。
- 2.6 节“创建向量”。

# 2.9　取得索引处的值

作者：Luke VanderHart

## 问题

有一个向量，需要取得特定位置（索引）的值。

## 解决方案

有几种实现方法。

### 使用 nth

`nth` 函数可用于所有序列，在向量这样的索引集合上使用时进行了特殊处理，能够提供常数访问时间：

```
(nth [:a :b :c :d] 2)
;; -> :c
```

如果给出的索引值超过向量的长度，`nth` 将抛出异常，除非提供可选的第 3 个参数，它会在索引越界时返回：

```
(nth [:a :b :c] 4)
;; -> IndexOutOfBoundsException

(nth [:a :b :c] 4 :not-found)
;; -> :not-found
```

### 将向量作为其索引的函数

向量本身也是函数，在用整数参数调用时，将返回该索引处的值：

```
(def v [:a :b :c])
(v 2)
;; -> :c
```

使用越界的索引来调用向量函数，将导致 `IndexOutOfBoundsException`。

### 使用 get

因为向量支持 Associative 接口，以整数索引为键，所以可以用 `get` 函数来取得索引处的值：

```
(get [:a :b :c] 2)
;; -> :c
```

不像 `nth`，如果向 `get` 提供越界的索引，它将返回 `nil`，而不是抛出异常，除非提供默认值，在键（这里就是索引）未找到时返回它：

```
(get [:a :b :c] 5)
;; -> :nil

(get [:a :b :c] 5 :not-found)
;; -> :not-found
```

## 讨论

应该使用哪种技术？这些方法都很好，你的选择取决于你如何看待向量。nth 专注于它的序列本质，而 get 强调了它是索引的、关联的。将向量作为一个函数也没错，Clojure 中所有关联集合都是其键的函数。

最后，在作出这种选择时，应该考虑以下问题。

- 哪种方法让代码更明白易懂？
- 你要处理的数据的本质是什么？例如，它是否看起来是个向量，而本质是一个序列（使用 nth）？或者本质是与索引关联的值（使用 get）？
- 备选技术的失效模式是什么？例如，返回 nil 或抛出异常是期望的结果吗？

## 参阅

- 2.16 节“从映射表中取得值”。

# 2.10 设置索引处的值

作者：Luke VanderHart

## 问题

对于给定的向量，需要得到一个新向量，在特定索引处具有不同的值。

## 解决方案

使用 assoc 来设置特定索引处的值。

```
(assoc [:a :b :c ] 1 :x)
;; -> [:a :x :c]
```

assoc 也可以同时设置多个索引处的值，只要提供额外的索引 / 值对。

```
(assoc [:a :b :c ] 1 :x 2 :y)
;; -> [:a :x :y]
```

## 讨论

你可能注意到，assoc 函数也用于设置映射表中键所对应的值。这是因为向量和映射表一样都是关联的，都实现了同样的接口（clojure.lang.Associative），这是 assoc 在背后用到的东西。

但与映射表不同，在 assoc 用于向量时，键必须是整数索引，并在向量的范围之内。试图使用非整数的键，将导致 IllegalArgumentException，试图使用超过向量长度的索引，将导致 IndexOutOfBoundsException。

注意，在使用 assoc 时，索引可以等于当前向量的长度（比最大的索引大 1）。这将导致元素添加在末尾。

## 参阅

- 2.7 节“在向量中‘添加’一个元素”。
- 2.18 节“设置映射表中的键。”

# 2.11 创建集

作者：Luke VanderHart

## 问题

需要创建由不同对象构成的、未排序的集合，能快速地检查元素是否属于该集合。

## 解决方案

使用集的字面值来创建对象集：

```
#{:a :b :c}
;; -> #{:a :c :b}

;; 集的字面值中的重复元素是错误
#{:x :y :z :z :z}
;; -> IllegalArgumentException Duplicate key: :y :z ...
```

使用 hash-set，利用参数创建集：

```
(hash-set :a :b :c)
;; -> #{:a :c :b}

(apply hash-set :a [:b :c])
;; -> #{:a :c :b}
```

使用 set，从其他集合创建集：

```
(set "hello")
;; -> #{\e \h \l \o}
```

或者利用 into 和一个集来创建一个新集：

```
(into #{} [:a :b :c :a])
;; -> #{:a :b :c}

(into #{:a :b} [:b :c :d])
;; -> #{:a :b :c :d}
```

**集构建的性能**

在本书编写时，对于大型的对象集合，into 方法比 set 方法要快 3 倍。如果要处理大型集合，又比较关注性能，请使用 into：

```
(def largeseq (doall (range 1e5)))

(time (dotimes [_ 100] (set largeseq)))
;; *out*
;; "Elapsed time: 5594.961 msecs"

(time (dotimes [_ 100] (into #{} largeseq)))
;; *out*
;; "Elapsed time: 1329.66 msecs"
```

用 sorted-set 创建一个排序的集：

```
(sorted-set 99 4 32 7)
;; -> #{4 7 32 99}

(into (sorted-set) "the quick brown fox jumps over the lazy dog")
;; -> #{\space \a \b \c \d \e \f \g \h \i \j \k \l \m \n \o \p
;;      \q \r \s \t \u \v \w \x \y \z}
```

# 讨论

集是很有用的数据结构。如果有一个值的集合，但只关心其中不同的值，常常会用到集。在集中查找元素的时间复杂度通常是 $O(1)$。

前面展示的技术构建的都是*散列集*，它们没有排序，使用散列表作为内部表示形式。

Clojure 也支持创建*排序集*，它维持元素的次序。利用 compare、sorted-set 创建的集保持元素升序排列。这在将集作为序列时是有用的：

```
(def alphabet (into (sorted-set) "qwertyuiopasdfghjklzxcvbnm"))
(last alphabet)
;; -> \z
(second (disj alphabet \b))
;; -> \c
```

排序集中的所有元素必须彼此进行比较（也就是说，排序集中不能既有字符串，又有数字）。如果添加不可比较的值，将导致运行时错误。

在排序集中添加和移除对象，总是会得到另一个排序集。

如果希望存储的值没有自然的排序次序（或者不想用它们的自然次序），可以用 sorted-set-by 来指明定制的比较方法。在添加或移除对象时，创建集时使用的比较方法将仍然有效：

```
(def descending-set (sorted-set-by > 1 2 3))

(into descending-set [-1 4])
;; -> #{4 3 2 1 -1}
```

选择散列集还是排序集，需要考虑性能上的妥协。散列集基于散列表，大多数情况下插入和查找时间是常数。但是，它们需要一定程度的内容消费。排序集基于平衡的红黑树，内存使用上更有效率，但查找和插入比较慢。

## 参阅

- 2.6 节“创建向量”。
- 2.12 节“在集中添加和移除元素”。

# 2.12 在集中添加和移除元素

作者：Luke VanderHart

## 问题

要添加或删除一些元素，得到一个新集。

## 解决方案

conj 函数支持集，就像它支持列表、向量和映射表一样。用它在集中添加元素，将集与要添加的任意多个元素传递给它：

```
(conj #{:a :b :c} :d)
;; -> #{:a :c :b :d}

(conj #{:a :b :c} :d :e)
;; -> #{:a :c :b :d :e}
```

要移除一个或多个元素，请使用 disj 函数，它是专门针对集的。它接受一个集，以及要移除的一个或多个键：

```
(disj #{:a :b :c} :b)
;; -> #{:a :c}
```

```
(disj #{:a :b :c} :b :c)
;; -> #{:a}
```

## 讨论

由于集是未排序的，就没有添加或移除的元素在“哪里”的概念。集要么包含一个元素，要么不包含。

注意，conj 和 disj 返回的集，与原来的集都有同样的具体类型。散列集还是散列集，排序集还是排序集。

另外值得注意的是，如果集中已包含或不包含要添加或移除的元素，这些操作就什么也不做。如果集已包含该元素，conj 将返回同样的集。如果指定的元素不在集中，disj 也返回同样的集。

如果要在集中添加或移除大量的元素，就应该考虑使用 clojure.set 命名空间中专门的集操作函数，尤其是 clojure.set/union（用于将多个集中的元素合并）和 clojure.set/difference（用于得到不包含在另一个集中的元素）。与多次调用或用大量参数来调用 conj 和 disj 相比，它们表示集操作通常更自然。

## 参阅

- 2.3 节“在列表中‘添加’一个元素”。
- 2.7 节“在向量中‘添加’一个元素”。
- 2.14 节“使用集操作”。

# 2.13　测试集成员

作者：Luke VanderHart

## 问题

要检查一个元素是不是集的成员。

## 解决方案

检查单个元素最容易的方法是 contains? 函数，它接受一个集和一个元素，如果集包含该元素，就返回 true：

```
(contains? #{:red :white :green} :blue)
;; -> false
```

```
(contains? #{:red :white :green} :green)
;; -> true
```

get 函数也适用于集，做的事情基本上一样，但它不返回 true 或 false。如果是成员，它就返回该值，不是就返回 nil：

```
(get #{:red :white :green} :blue)
;; -> nil

(get #{:red :white :green} :green)
;; -> :green
```

最后，集也是函数。用一个参数去调用时，它像 get 一样，如果是成员就返回该参数，否则返回 nil：

```
(def my-set #{:red :white :green})

(my-set :blue)
;; -> nil

(my-set :green)
;; -> :green
```

还要注意，关键字对集的行为与对映射表的行为一样。因此，下面与使用 get 是等价的：

```
(:blue #{:red :white :green})
;; -> nil

(:green #{:red :white :green})
;; -> :green
```

## 讨论

选择 contains 还是 get 主要是一种美学考量。但是，如果在集中有可能包含 nil 这个至关重要的值，就肯定要用 contains?，因为 get 返回 nil 时，不能告诉你任何信息。

用集作为函数很有趣，而且如果用它作为序列操作的谓词函数，还非常有用。例如，常常需要过滤一个序列，让它只包含集中的元素。在这种情况下，用集本身作为函数既方便，又符合习惯：

```
(take 10
  (filter #{1 2 3}
         (repeatedly #(rand-int 10))))
;; -> (2 1 2 3 2 2 1 2 2 1)
```

这段代码先利用 repeatedly 反复调用 rand-int（包装成一个匿名函数），创建了一个无限惰性序列，包含 1 至 10 的随机数。然后让这个序列通过一个过滤器，包含 1 至 3 的集作

为过滤谓词。结果是另一个无限惰性序列，但只包含谓词集中的成员。

这个例子是编造的。但是，集作为谓词函数是非常有用的技巧，经常出现在 Clojure 项目中。

## 参阅

- 2.14 节“使用集操作”。
- 2.16 节“从映射表中取得值”。

# 2.14 使用集操作

作者：Luke VanderHart

## 问题

需要在集上执行常见操作，如并集、交集和差集，或测试一个集是不是另一个集的子集或超集。

## 解决方案

以下这些函数都是 Clojure 内建支持的，在 `clojure.set` 命名空间中提供。

`union` 接受任意多个集作为参数，返回它们的并集（该集包含所有集的所有元素）：

```
(clojure.set/union #{:red :white} #{:white :blue} #{:blue :green})
;; -> #{:white :red :blue :green}
```

`intersection` 接受任意多个集作为参数，返回它们的交集（该集只包含所有集中都有的元素）：

```
(clojure.set/intersection #{:red :white :blue}
                          #{:red :blue :green}
                          #{:yellow :blue :red})
;; -> #{:red :blue}
```

`difference` 接受一组集作为参数，返回集中的元素在第一个集中，而不在后续集中：

```
(clojure.set/difference #{:red :white :blue :yellow}
                        #{:red :blue}
                        #{:white})
;; -> #{:yellow}
```

如果 `subset?` 的第一个参数是第二个参数的子集，它将返回 `true`（也就是说，第一个集中的所有元素都在第二个集中）：

```
(clojure.set/subset? #{:blue :white}
                     #{:red :white :blue})
;; -> true

(clojure.set/subset? #{:blue :black}
                     #{:red :white :blue})
;; -> false
```

superset? 的工作方式与 subset? 相同，不同之处在于，当第一个集是第二个集的超集时，它返回 true。

你也许注意到，superset? 和 subset? 实际上是等价的，只是参数的次序颠倒了。

## 讨论

一般来说，如果适用，就应该尝试使用这些集操作函数。集代表了一大部分数据，大多数开发者每天都要面对它们，不论这些数据是否被看作集，或显式地建模为集。

大量的缺陷源于对集合行为的假定。在编程时，出于某个目的使用了某种数据结构类型，这实际上是一种*信息传递*，从代码的最初作者传递给将来的程序员，说明了关于集合本质的一些事情。集是*未排序的*、*唯一的*集合，它们强调的重点是元素是否属于集，而不是元素的次序或出现的次数。

如果数据确实代表一个逻辑集，那就可以用集这种数据结构来建模，然后尝试以集操作的方式来考虑它的操作。在很多情况下，你会发现这让程序更容易解释，程序本身更像文档，说明了集中所包含的数据的来源和用途。

## 参阅

- 2.13 节“测试集成员”。
- 2.26 节“检测集合是否包含几个值中的一个”。

# 2.15 创建映射表

作者：Luke VanderHart

## 问题

要创建一个关联，映射键与值。可能需要维持键的特定次序。

## 解决方案

利用映射表字面值（花括号），用不同的键和值来创建简单的映射表：

```
{:name ""
 :class :barbarian
 :race :half-orc
 :level 20
 :skills [:bashing :hacking :smashing]}
```

键和值可以是任意类型。如果其结构一眼看上去难以区分，可以用逗号作为键值对的分隔符：

```
{1 1, 8 64, 2 4, 9 81}
```

在 Clojure 中，逗号是空白符，这意味着它们可以用在任何地方，对值没有影响。它只是让代码更易读的一种方法。

要创建空的、未排序的映射表，就用一对花括号：`{}`。

要创建具体类型的映射表，请使用映射表构造函数。`array-map`、`hash-map` 和 `sorted-map` 各自返回对应类型的映射表：

```
(array-map)
;; -> {}

(sorted-map :key1 "val1" :key2 "val2")
;; -> {:key1 "val1" :key2 "val2"}
```

如果一个键在参数列表中出现多次，最终返回的映射表将采用最后一个值。

使用 `sorted-map-by` 函数，利用定制的比较符来创建排序的映射表：

```
(sorted-map-by #(< (count %1) (count %2))
               "pigs" 14
               "horses" 2
               "elephants" 1
               "manatees" 3)
;; -> {"pigs" 14, "horses" 2, "manatees" 3, "elephants" 1}
```

# 讨论

Clojure 映射表有三种具体实现。

- 数组映射表，`clojure.lang.PersistentArrayMap`
  背后是简单的数组。它们对很小的映射表是有效率的，但对于较大的映射表效率不高。

- 散列映射表，`clojure.lang.PersistentHashMap`
  背后是散列表数据结构。散列表支持近似常数时间的查找和插入，但也要求一定的空间

开销，使用的堆空间要多一点。

- 排序映射表，clojure.lang.PersistentTreeMap
  背后是平衡红黑树。它们比散列映射表的空间效率更高，但插入和访问的时间要慢一些。

数组映射表是小映射表的默认实现（在本书写作时，是指少于 10 个条目），散列映射表是较大映射表的默认实现。排序映射表只能通过调用 `sorted-map` 或 `sorted-map-by` 函数来创建。

对排序映射表使用 `assoc` 或 `conj`，将得到另一个排序映射表。但是，对数组映射表使用 `assoc` 时，如果达到一定大小，就会返回散列映射表。反过来是不成立的，对散列映射表使用 `dissoc`，不会得到数组映射表，即使它变得非常小。

## 参阅

- 1.29 节“比较日期”，以及 1.17 节“模糊比较”，探讨了更多比较的用法。
- 2.11 节“创建集”。
- 2.18 节“设置映射表中的键”。
- 2.24 节“值的比较与排序”。

# 2.16　从映射表中取得值

作者：Luke VanderHart

## 问题

需要取得映射表中特定键对应的值。

## 解决方案

对于映射表，有几种方法取得键对应的值。

最直接的方式是使用 `get` 函数，它对给定的映射表和键，返回键对应的值，如果映射表不包含该键，就返回 `nil`：

```
(get {:name "Kvothe" :class "Bard"} :name)
;; -> "Kvothe"

(get {:name "Kvothe" :class "Bard"} :race)
;; -> nil
```

在需要时，也可以传入第三个参数，作为默认返回值。如果映射表不包含该键，就会返回

这个值，而不是 nil：

```
(get {:name "Kvothe" :class "Bard"} :race "Human")
;; -> "Human"
```

如果映射表使用关键字作为键，可以用关键字本身作为函数。关键字实现了 IFn 接口，以映射表作为参数调用时，它们会在映射表中查找自己，如果存在就返回值，否则返回 nil。也可以提供第二个参数作为默认值，在查找不到时返回，就像 get 那样：

```
(:name {:name "Marcus" :class "Paladin"})
;; -> "Marcus"

(:race {:name "Marcus" :class "Paladin"} "Human")
;; -> "Human"
```

最后，在映射表中查找值的第三种方法，就是用映射表本身作为函数，将要查找的键作为参数。就像 get 和关键字函数一样，也可以传入第二个参数作为默认值，在键查找不到时返回，否则会返回 nil：

```
(def character {:name "Brock" :class "Barbarian"})

(character :name)
;; -> "Brock"

(character :race)
;; -> nil

(character :race "Human")
;; -> "Human"
```

要查找嵌套的映射表，有一个方便的函数：get-in。不是传入一个键，而是传入键的序列，它会连续查找嵌套的结构，就像在嵌套结构的每一层上反复调用 get。未找到就返回 nil：

```
(get-in {:name "Marcus" :weapon {:type :greatsword :damage "2d6"}}
        [:weapon :damage])
;; -> "2d6"

(get-in {:name "Marcus"}
        [:weapon :damage])
;; -> nil
```

get-in 也接受可选的默认值，如果嵌套层级中有任何键未找到，就会返回它：

```
(get-in {:name "Marcus"}
        [:weapon :damage]
        "1d2 (fists)")
;; -> "1d2 (fists)"
```

注意，get-in 适用于所有关联数据结构，不只是映射表。这意味着它可以组合使用，例如，与向量的索引一起使用：

```
(get-in [{:name "Marcus" :class "Paladin"}
         {:name "Kvothe" :class "Bard"}
         {:name "Felter" :class "Druid"}]
        [1 :class])
;; -> "Bard"
```

## 讨论

这三种技术哪种最好？它们的语义是一样的，但在 Clojure 的使用习惯中，它们传达了不同的暗示，说明了它们的适用范围。

通常，用关键字作为函数来查找，表示映射表被作为“对象”，键作为“字段”，其中映射表包含相对较少的、众所周知的一组键，并且可以合理地假设，键确实存在。

get 函数和把映射表作为函数的查找技术则不同，它们常用于较大的映射表，可能的键的集合通常是开放的。在这两者之间选择没有太多差别，唯一的区别是需要注意，如果出于某种原因，提供的映射表是 nil，将它用作函数就会抛出异常，而对 nil 使用 get 就会返回 nil。

同时也值得注意，用映射表本身作为函数，这不仅仅是一种为了简便的随意行为。从“函数”这个词的技术意义上说，映射本身就是键到值的函数。请考虑下面的函数定义和映射表：

```
(defn square [x] (* x x))

(def square {1 1, 2 4, 3 9, 4 16, 5 25})
```

使用 (square 3) 的调用形式时，调用者实际上*不知道* square 是“真正的”函数还是映射表。当然，正常定义的函数在这个例子中有一些优势。比如说，它的定义域没有限制，不只是列出来的那些键。乘法函数也相当快，所以事先计算的结果没有优势。但在某些情况下，有些函数确实有自然约束的定义域，而且计算代价更大，能用函数的映射表实现，可以带来真实的性能提升。

在从映射表中取值的所有不同技术中，如果键不存在，都会返回 nil，所以需要一些特殊处理，才能区分键存在但值为 nil 以及键根本不存在的情况。

最容易的办法就是，总是提供默认值，在找不到键时返回。要绝对确保能够区分默认值和映射表中可能包含的值，可以使用命名空间限定的关键字（如 ::not-found）。

也可以使用 contains? 函数，它接受集合和键作为参数，当且仅当集合中包含该键时（即使值为 nil），才会返回 true。

**contains? 的含义**

contains? 函数的确切行为常常导致困惑，因为其他许多语言都有类似名称的函数，做的事却不一样。

在 Clojure 中，contains? 不是一个查找函数，它不会通过检查集合找出某个元素。准确地说，它是一个查表函数，只适用于关联或索引集合。在其他语言中，它常被命名为 containsKey 或类似的名称。

这意味着它可以应用于映射表和集，如果指定的键是有效的键，或是集中的成员，它将返回 true。但对于向量，只有传入的整数在 0 和向量的最大索引之间时，它才返回 true。如果用于列表或序列，它将抛出异常。

## 参阅

- 2.17 节“从映射表中同时取出多个键”。
- 2.18 节“设置映射表中的键”。
- 2.20 节“将映射表作为序列（或反过来）”。

# 2.17 从映射表中同时取出多个键

作者：Leonardo Borges

## 问题

需要同时从映射表中取出多个值。

## 解决方案

如果返回值的次序不重要，就用 vals 和 select-keys：

```
;; 红豆和绿豆共有多少?
(def beans {:red 10
            :blue 3
            :green 1})

(reduce + (vals (select-keys beans [:red :green])))
;; -> 11
```

如果次序重要，就用 juxt：

```
;; 红豆和绿豆分别有多少?
((juxt :red :green) beans)
;; -> [10 1]
```

## 讨论

对于从映射表中取出多个键对应的值，juxt 以及组合使用 vals 和 select-keys 都是适合的工具。但它们的行为有一些细微差别，理解这些差别很重要。

初看上去，juxt 的方式似乎明显胜出。但仅限于初看：如果要取的键有一个不是关键字（更准确地说，不是函数），这种方法就会崩溃。这是因为 juxt 只是并列多个函数的返回值。因为关键字是函数，所以可以 juxt 它们，得到严格排序的值列表。

如果 beans 映射表中的键是字符串，就不可能用 juxt 来取得它们的值：

```
((juxt "a" "b") beans)
;; -> ClassCastException java.lang.String cannot be cast to clojure.lang.IFn ...
```

但是，select-keys 能够取得多个任意键的值。select-keys 接受映射表和键的序列作为参数，返回一个新映射表，其中仅包含这些键和对应的值：

```
(def weird-map {"a" 1, {:foo :bar} :baz, 13 31})

(select-keys weird-map
             ["a" {:foo :bar}])
;; -> {{:foo :bar} :baz, "a" 1}

(vals {{:foo :bar} :baz, "a" 1})
;; -> (1 :baz)
```

因为映射表不是排序的，所以假定键和值的次序不变是不安全的（即使偶然遇到的例子中是这样）。如果从非关键字的映射表中取得多个值，最容易的方法可能是用 juxt 将这种交互包装起来：

```
(def a-str-then-foo-bar-map
  (juxt #(get % "a")
        #(get % {:foo :bar})))

(a-str-then-foo-bar-map weird-map)
;; -> [1 :baz]
```

现在你应该能避开古怪的映射表了吧？

## 参阅

- 2.16 节“从映射表中取得值”。
- 2.19 节“用复合值作为映射表的键”。

# 2.18 设置映射表中的键

作者：Luke VanderHart

## 问题

需要“更改”映射表，添加、设置或移除一些键。

## 解决方案

更改映射表最基本的方法是用 assoc 函数。它接受映射表和任意数目的键值对作为参数，返回更新的映射表，包含相应的键和值。

```
(def villain {:honorific "Dr." :name "Mayhem"})
(assoc villain :occupation "Mad Scientist" :status :at-large)
;; -> {:honorific "Dr.", :name "Mayhem",
;;     :occupation "Mad Scientist", :status :at-large}
```

如果 assco 函数用于已经包含某个键的映射表，它将返回更新过的映射表，该键将对应新指定的值：

```
(def villain {:honorific "Dr.", :name "Mayhem",
              :occupation "Mad Scientist", :status :at-large})
(assoc villain :status :deceased)
;; -> {:honorific "Dr.", :name "Mayhem",
;;     :occupation "Mad Scientist", :status :deceased}
```

要移除一些键，就用 dissoc 函数。它接受映射表和任意数目的键作为参数，返回除去这些键的映射表：

```
(def villain {:honorific "Dr.", :name "Mayhem",
              :occupation "Mad Scientist", :status :deceased})
(dissoc villain :occupation :honorific)
;; -> {:name "Mayhem", :status :deceased}
```

## 讨论

映射表中包含其他映射表是很常见的。如果必须更新深度嵌套的值，嵌套调用 assoc 很快就会变得不方便，尤其是因为它们需要“从内到外”。考虑下面的数据结构：

```
(def book {:title "Clojure Cookbook"
           :author {:name "Ryan Neufeld"
                    :residence {:country "USA"}}})
```

如果 Ryan 回到了他的故乡加拿大，要只用 assoc 来更新表示这本书的映射表，看起来就是这样：

```
(assoc book :author
  (assoc (:author book) :residence
    (assoc (:residence (:author book)) :country "Canada")))
```

显然，这不方便，也难以阅读。

`assoc-in` 函数消除了这种不便，它允许指定*键路径*，而不是单独一个键。*键路径*列出了键的序列，递归地应用，改变深度嵌套的值，而不是只改变单个键深度的值。

```
(assoc-in book
          [:author :residence :country]
          "Canada")
;; -> {:author {:name "Ryan Neufeld"
;;              :residence {:country "Canada"}}
;;              :title "Clojure Cookbook"}
```

前面的例子首先在嵌套的数据结构中，查找与 `:residence` 键关联的映射表，然后将 `"Canada"` 与 `:country` 键关联。最后，返回整个数据结构。

如果需要基于以前的值来*更新*它，而不只是改变它呢？

幸运的是，Clojure 提供了 `update-in`，专门用于这一目的。`update-in` 不是接受一个新值，而是接受一个*更新函数*。这个函数被调用，参数是*键路径*所取得的值，以及传给 `update-in` 的后续所有参数。这初看起来是一个奇怪的函数。也许最好用例子来说明：

```
(def website {:clojure-cookbook {:hits 1236}})

;; 记录 Cookbook 网站的 101 次新点击
(update-in website                 ; ❶
           [:clojure-cookbook :hits] ; ❷
           +                       ; ❸
           101)                    ; ❹
;; -> {:clojure-cookbook {:hits 1337}}
```

❶ 映射表。

❷ 键路径。

❸ 更新函数，`+`。

❹ `+` 的其他参数。

如果向量中的键不存在，`update-in` 也会为这些键创建映射表。这意味着它可以用于创建结构，同时更新值：

```
(update-in {} [:author :residence] assoc :country "USA")
;; -> {:author {:residence {:country "USA"}}}
```

即使开始映射表是空的，也会为 `:author` 和 `:residence` 键创建两个空的映射表，这意味着 `assoc` 将应用于一个新的空映射表。

### 像映射表一样对待 Clojure 的状态结构

映射表的另一种常见用法，是作为 Clojure 状态结构的值，这些结构是 atom、ref 或 agent。Clojure 映射表的值不是固定不变的。从字面来讲，如果你向映射表“增加”一个键，其值就会发生变化。但有时为不同的值保存逻辑的身份也是有必要的。那就牵扯到何时使用状态管理工具的问题。

要更新一个状态的值（ref、atom 或 agent），就要调用其具体的转换函数（分别是 `alter`、`swap!` 和 `send`）。状态转换函数具有同样的形式：它们接受引用作为第一个参数，要作用于值的函数作为第二个参数，该函数其他的参数作为附加的参数。

所以，举例来说，要深度更新映射表中的一个元素，该映射表由一个 atom 引用，就可以调用 `swap!` 函数（atom 的状态转换函数），将 atom 和 `update-in` 函数传递给它，再加上键的列表和用于更新该值的函数：

```
(def retail-data (atom {:customers [{:id 123 :name "Luke"}
                                     {:id 321 :name "Ryan"}]
                        :orders [{:sku "Q2M9" :customer 123 :qty 4}
                                 {:sku "43XP" :customer 321 :qty 1}]}))

(swap! retail-data update-in [:orders] conj
       {:sku "9QED" :customer 321 :qty 2})
```

这会在订单列表中添加一个新的订单映射表，订单列表在一个映射表中，映射表在 `retail-data` atom 中。

尽管这种三重组合不太常见，但它们说明了函数的总体一致性，即接受其他函数和参数作为参数，也说明了它们能够实现任意深度的嵌套。在这个例子中，从调用 `swap!` 开始，结果也以同样的方式更新了映射表，连结了向量。

## 参阅

- 2.20 节“将映射表作为序列（或反过来）”。
- 2.22 节“一个键保存多个值”。
- 2.23 节“合并映射表”。

# 2.19　用复合值作为映射表的键

作者：Luke VanderHart

## 问题

要用一个值作为映射表中的查找键，但它不是简单的原生类型。例如：

- 要用地理坐标或笛卡儿坐标作为映射表的键；
- 希望关联一些值和一些函数。

## 解决方案

因为 Clojure 对复合值有强大的标识语义，所以完全支持用任何不变的值作为映射表的键。更重要的是，这样做效率更高。

例如，考虑代表国际象棋棋盘的数据结构，8 × 8 的格子，每个位置可以放六种棋子中的一种。行用数字 1 至 8 表示，列用字母 a 至 h 表示。

在 Clojure 中，可以将它直接表示为映射表：

```
(def chessboard {[:a 5] [:white :king]
                 [:a 4] [:white :pawn]
                 [:d 4] [:black :king]})
```

移动棋子需要两个操作，`dissoc` 老的位置，`assoc` 新的位置：

```
(defn move
  "Given a map representing a chessboard, move the piece at src
  to dest"
  [board source dest]
  (-> board
      (dissoc source)
      (assoc dest (board source))))

(move chessboard [:a 5] [:a 4])
;; -> {[:d 4] [:black :king]
;;     [:a 4] [:white :king]}
```

作为非传统映射表键的另一个例子，请考虑这种情况：有一组函数，要能够指定每个函数的“权重”，并在函数被调用时，用返回值乘以对应的权重。

一种容易的实现方法是将函数和权重保存在映射表中，以函数作为键：

```
(def plus-two (partial + 2))
(def plus-three (partial + 3))
(def weight-map {plus-two 1.0
                 plus-three 0.8})
```

然后用一个简单的包装函数，调用函数以及适用的权重：

```
(defn apply-weighted
  "Given a weight map, a function, and args, applies the function
  to the args, multiplying the result by the weighting for the
  function. If the weight map does not specify a weight for the
  function, a default of 1.0 is used."
  [weight-map f & args]
```

```
  (* (get weight-map f 1.0)
     (apply f args)))

(apply-weighted weight-map plus-two 2)
;; -> 4.0

(apply-weighted weight-map plus-three 1)
;; -> 3.2
```

## 讨论

使用两维数组是象棋游戏建模的传统方式，在 Clojure 中，就是用向量的向量。

这样做肯定是合理的，而且（可能）性能好一点点。但是，这个模型对现实问题来说没那么清晰。比方说，它需要将象棋的行列数字和字母翻译成向量的索引。使用映射表能直接存储位置，用象棋本身的术语。

类似地，函数权重的例子也有另外的实现方式。可以用 `cond` 语句来实现，列出所有函数和权重，或者用一个协议方法（protocol method）取代所有的函数，然后有带不同权重的各种实现。

但是，将函数和权重保存在映射表中，好处是一眼就能看出某个函数的权重是多少。更重要的是，有可能保存多组不同的权重，运行时动态更换不同的权重策略。

## 参阅

- 2.16 节“从映射表中取得值”，2.18 节“设置映射表中的键”。

# 2.20　将映射表作为序列（或反过来）

作者：Luke VanderHart

## 问题

需要将映射表的内容作为条目序列。或者，需要将条目序列转成映射表。

## 解决方案

要得到映射表的序列视图，只要对它调用 `seq`。注意，大多数序列处理函数自己会在参数上调用 `seq`，所以通常不需要显式地调用 `seq`：

```
(seq {:a 1, :b 2, :c 3, :d 4})
;; -> ([:a 1] [:c 3] [:b 2] [:d 4])
```

这创建了一个键值对序列，然后可以像处理其他序列一样处理它了。

要从序列创建映射表，可以利用 conj 函数的一个特点，即它在应用于映射表时，可以接受两个元素的向量，作为键值对，将其添加到映射表中：

```
(def m {:a 1, :b 2})
(conj m [:c 3])
;; -> {:a 1, :b 2, :c 3}
```

因为 into 函数反复应用 conj，将元素从一个序列放到一个集合中，所以可以用它将键值对序列转换成一个映射表：

```
(into {} [[:a 1] [:b 2] [:c 3]])
;; -> {:a 1, :b 2, :c 3}
```

也可以从两个序列构造一个映射表，一个包含键，另一个包含值。这就是 zipmap 函数的功能。给定两个序列，它将返回一个映射表，键来自第一个参数序列，值来自第二个参数序列：

```
(zipmap [:a :b :c] [1 2 3])
;; -> {:c 3, :b 2, :a 1}
```

传递给 zipmap 的两个序列，如果一个比较短，多余的值就会被忽略，得到的映射表和较短的序列一样长。

## 讨论

在获取*散列映射表*的序列视图时，返回的映射表项的次序是任意的，或未定义的。也有一点便利，如果同一个映射表多次转换成序列，这个次序（尽管任意）*是*确保一致的。

在使用*排序映射表*时，条目返回的次序是它们在映射表中的排序次序。例如：

```
(seq (hash-map :a 1, :b 2, :c 3, :d 4))
;; -> ([:a 1] [:c 3] [:b 2] [:d 4])

(seq (sorted-map :a 1, :b 2, :c 3, :d 4))
;; -> ([:a 1] [:b 2] [:c 3] [:d 4])
```

关于这个序列中的条目值，还有一个有趣的事实。它们被当作向量输出，而且它们就是向量，因为它们实现了完整的向量接口。但是它们的具体类型实际上不是 clojure.lang.PersistentVector，而是一种不同类型的向量，名为*映射表条目*，它不仅是向量，也支持 clojure.lang.MapEntry 接口。

MapEntry 接口提供了 key 和 val 函数，用于取得条目的键和值：

```
(def entry (first {:a 1 :b 2}))

(class entry)
```

```
;; -> clojure.lang.MapEntry

(key entry)
;; -> :a

(val entry)
;; -> :1
```

在将映射表作为序列时，对映射表条目应该优先选择这些函数，而不是 first 和 second 函数，因为它们保留了键值对的语义，让代码更容易阅读。

## 参阅

- 2.21 节“对映射表应用函数”。

# 2.21 对映射表应用函数

作者：Luke VanderHart

## 问题

要对映射表的键或值应用转换函数。

## 解决方案

使用下面简单的通用函数，改变参数来满足各种需要：

```
(defn map-keys
  "Given a map and a function, returns the map resulting from applying
  the function to each key."
  [m f]
  (zipmap (map f (keys m)) (vals m)))

(map-keys {"a" 1 "b" 2} keyword)
;; -> {:b 2, :a 1}

(defn map-vals
  "Given a map and a function, returns the map resulting from applying
  the function to each value."
  [m f]
  (zipmap (keys m) (map f (vals m))))

(map-vals {:a 1, :b 1} inc)
;; -> {:b 2, :a 2}

(defn map-kv
  "Given a map and a function of two arguments, returns the map
  resulting from applying the function to each of its entries. The
```

```
    provided function must return a pair (a two-element sequence.)"
    [m f]
    (into {} (map (fn [[k v]] (f k v)) m)))

(map-kv {"a" 1 "b" 1} (fn [k v] [(keyword k) (inc v)]))
;; -> {:a 2, :b 2}
```

## 讨论

map-keys 和 map-vals 非常简单明了。它们开始都用 keys 和 vals 函数，将映射表 m 分解成键的序列和值的序列。然后它们利用 map 函数，转换键序列或值序列。最后，用 zipmap 函数将键序列和值序列合并成一个映射表，更新就完成了。

map-kv 有点不一样。它开始将映射表转换成映射表条目的序列，然后利用 map，对它们应用一个匿名函数，解构键和值，然后将键和值传给调用者提供的函数。最后，用 into 反复将得到的键值对连接到一个空映射表中，返回新的映射表，包含转换过的键和值。

下面的例子是等价的，但没有用解构，所以高层结构更清楚一些：

```
(defn map-kv
  [m f]
  (into {} (map (fn [entry]
                  (f (key entry) (val entry)))
                m)))
```

显而易见，这三个函数都是标准 map 函数应用于映射表数据结构的即兴发挥。另一个函数式编程的主力 reduce 呢？

Clojure 已经自带了 reduce-kv 函数，是在 1.4 版中添加的。reduce-kv 接受三个参数：一个函数、一个初始值，和一个关联集合。提供的函数也必须接受 3 个参数。reduce-kv 对提供的集合进行归约，先将函数应用于初始值、映射表的第一个键和值，再是结果值、第二个键和值，再是结果值、第三个键和值，以此类推。

下面的例子用 reduce-kv 来得到映射表中所有值之和：

```
(reduce-kv (fn [agg _ val]
             (+ agg val))
           0
           {:a 1 :b 2 :c 3})
;; -> 6
```

注意，在函数的参数声明中，下划线（_）代替了 key。这是 Clojure 中的习惯用法，命名函数体中实际上不会用到的参数。

也可以用 reduce-kv 重新定义 map-kv：

```
(defn map-kv
    [m f]
    (reduce-kv (fn [agg k v] (conj agg (f k v))) {} m))
```

可以用在这个例子中：

```
(map-kv {:one 1 :two 2 :three 3}
        #(vector (-> %1 str (subs 1)) (inc %2)))
;; -> {"one" 2, "three" 4, "two" 3}
```

## 参阅

- 2.20 节“将映射表作为序列（或反过来）”。

# 2.22 一个键保存多个值

作者：Luke VanderHart

## 问题

通常，映射表中每个键严格地对应一个值：如果 assoc 一个已有的键，原来的值就会被替换。但是，有时候要用到类似映射表的接口（“多重映射表”），能在一个键下保存多个值。

需要在 Clojure 中创建一个类似映射表的数据结构，实现类似多重映射表的接口。

## 解决方案

为了在普通映射表的基础上引入这种能力，就要创建并扩展 MultiAssociative 协议，定义这种行为：

```
(defprotocol MultiAssociative
  "An associative structure that can contain multiple values for a key"
  (insert [m key value] "Insert a value into a MultiAssociative")
  (delete [m key value] "Remove a value from a MultiAssociative")
  (get-all [m key] "Returns a set of all values stored at key in a
                    MultiAssociative. Returns the empty set if there
                    are no values."))

(defn-value-set?
  "Helper predicate that returns true if the value is a set that
  represents multiple values in a MultiAssociative"
  [v]
  (and (set? v) (::multi-value (meta v))))

(defn value-set
  "Given any number of items as arguments, returns a set representing
  multiple values in a MultiAssociative. If there is only one item,
```

```
    simply returns the item."
    [& items]
    (if (= 1 (count items))
      (first items)
      (with-meta (set items) {::multi-value true})))

  (extend-protocol MultiAssociative
    clojure.lang.Associative
    (insert [this key value]
    (let [v (get this key)]
      (assoc this key (cond
                        (nil? v) value
                        (value-set? v) (conj v value)
                        :else (value-set v value)))))
  (delete [this key value]
    (let [v (get this key)]
      (if (value-set? v)
        (assoc this key (apply value-set (disj v value)))
        (if (= v value)
          (dissoc this key)
          this))))
  (get-all [this key]
    (let [v (get this key)]
       (cond
        (value-set? v) v
        (nil? v) #{}
        :else #{v}))))
```

当然，还有对应的单元测试（利用 clojure.test）：

```
(require '[clojure.test :refer :all])

(deftest test-insert
  (testing "inserting to a new key"
    (is (= {:k :v} (insert {} :k :v))))
  (testing "inserting to an existing key (single existing item)"
    (let [m (insert {} :k :v1)]
      (is (= {:k #{:v1 :v2}}
             (insert m :k :v2)))))
  (testing "inserting to an existing key (multiple existing items)"
    (let [m (insert (insert {} :k :v1) :k :v2)]
      (is (= {:k #{:v1 :v2 :v3}}
             (insert m :k :v3))))))

(deftest test-delete
  (testing "deleting a non-present key"
    (is (= {:k :v} (delete {:k :v} :nosuch :nada))))
  (testing "deleting a non-present value"
    (is (= {:k :v} (delete {:k :v} :k :nada))))
  (testing "deleting a single value"
    (is (= {} (delete {:k :v} :k :v))))
  (testing "deleting one of two values"
    (let [m (insert (insert {} :k :v1) :k :v2)]
      (is (= {:k :v1} (delete m :k :v2)))))
```

```
  (testing "deleting one of several values"
    (let [m (insert (insert (insert {} :k :v1) :k :v2) :k :v3)]
      (is (= {:k #{:v1 :v2}} (delete m :k :v3))))))

(deftest test-get-all
  (testing "get a non-present key"
    (is (= #{} (get-all {} :nosuch))))
  (testing "get a single value"
    (is (= #{:v} (get-all {:k :v} :k))))
  (testing "get multiple values"
    (is (= #{:v1 :v2} (get-all (insert (insert {} :k :v1) :k :v2) :k)))))

(run-tests)
;; -> {:type :summary, :pass 11, :test 3, :error 0, :fail 0}
```

# 讨论

首先，这段代码定义了一个协议来实现一组构成多重映射表的行为的函数。在这种情况下，协议是很好的选择：它将几个方法捆绑在一起，在同样的对象上执行相关的操作，并且允许多种具体实现。

在这个例子中，要实现期望的功能，需要三个方法。注意，协议的实现没有覆写或重新实现核心的映射表方法（`assoc`、`dissoc` 等）。它只是新行为的语义，区别于那些普通的映射表。围绕这些核心函数，Clojure 定义了非常强大的语义。破坏或覆写这些预期的语义总是不好的，尤其是如果使用一组不同的函数，能清楚地说明何时用到多重映射表的功能，覆写就更不好了。

`MultiAssociative` 的具体实现将该协议扩展到 `clojure.lang.Associative` 接口。肯定可以实现为更有针对性的东西，比如 `IPersistentSet`，但因为它实现时只*要求*有关联的东西，所以最好不要太具体。针对 `clojure.lang.Associative` 来编码也顺便提供了几种额外的能力。例如，现在自然有了“多重向量”，能够在每个索引处保存多个值（只要它们是通过 `insert` 添加的）。

阅读这段代码，你会注意到，许多逻辑实际上是要确保单值简单地存储，多重值包装在集中。这在插入和删除元素时得到保持，要求这些函数检查值的类型，相应进行包装或解包。类似地，`get-all` 需要将单值包装成集再返回，因为规定它必须返回集。

这是设计决定，有利有弊。也可以总是把值包装在集中，就算单值也这样。这会让代码简单一些，消除大多数的类型检查，以及形式的包装和解包。

但是，简单是有代价的。如果值（甚至单值）*总是*包装在集中，作为多重映射表的映射表就不容易通过普通的映射表函数来使用。它会包含许多看起来奇怪的单元素集，而且如果用 `assoc` 来添加元素，就会和将来对该键使用 `insert` 不兼容。

本质上，包装和解包是允许映射表既能用标准的 Associative 接口，也能用 MultiAssociative 接口，不要求用户记住它是“哪种”映射表。用 assoc 插入的值，可以用 get-all 读取，用 insert 插入的值，可以用 dissoc 移除。所有对普通的映射表的期望都是成立的。如果对多值的键调用普通的 get，将返回包含多个元素的集。这可能是用户检查数据时所期望的。

这段代码还有一点值得说明，它在保存多值的集上，使用了 ::multivalue 元数据，通过 value-set 和 value-set? 函数实现和测试。

这样做是为了处理一种特殊情况，即键对应的值本身是一个集。代码需要一种方法，来确认什么是为了管理多值而创建的集，什么是用户作为单值提供的集。

实现机制是在为多值而创建的集中加上元数据。使用一个命名空间限定的关键字，来确保它不会与用户提供的值的元数据发生冲突。然后，代码要做的就是检查集是否有 ::multivalue 元数据，就知道它是包含多值的集，还是本身就是值。

## 参阅

- 3.10 节“扩展内建的类型”。

# 2.23　合并映射表

作者：Tom Hicks

## 问题

需要将两个或多个映射表合并为一个。

## 解决方案

用 merge 来合并两个或多个没有相同键的映射表：

```
(def arizona-bird-counts {:cactus-wren 8})
(def florida-bird-counts {:gull 20 :pelican 14})
(merge florida-bird-counts arizona-bird-counts)
;; -> {:pelican 14, :cactus-wren 8, :gull 20}
```

如果多个映射表中有相同的键，需要明确控制合并的策略，就用 merge-with：

```
(def florida-bird-counts    {:gull 20 :pelican 1 :egret 4})
(def california-bird-counts {:gull 12 :pelican 4 :jay 3})

;; 用 + 合并值，求得和
(merge-with + california-bird-counts florida-bird-counts)
;; -> {:pelican 5, :egret 4, :gull 32, :jay 3}
```

## 讨论

对于 merge 和 merge-with，映射表的合并都是从左到右，返回不可变的新映射表。这些函数就像“左折叠”。merge 比较简单，总是返回看到的每个键的最后一个值。

如果同一个键的映射出现在多个映射表中，结果将采用后出现的映射。这可能是有用的，例如，如果在一天中多次收到新的总数，但只针对那些改变的值：

```
;; 一天中最喜爱的冰淇淋口味投票
(def votes-am {:vanilla 3 :chocolate 5})
(def votes-pm {:vanilla 4 :neapoliton 2})
(merge votes-am votes-pm)
;; -> {:vanilla 4, :chocolate 5, :neapoliton 2}
```

merge-with 为映射表合并提供了强大的能力，允许控制值合并的方式。可以将 merge-with 看成是 reduce 了多个映射表中同样的键。merge-with 的第一个参数是一个合并函数，供每一对复制的值调用。

通过仔细选择映射表中值的类型，merge-with 为一些常见问题提供了简明的解决方案。例如，通过带 clojure.set/intersection 的合并，可以找到团队中程序员“喜欢”和“不喜欢”的交集：

```
(def Alice {:loves #{:clojure :lisp :scheme} :hates #{:fortran :c :c++}})
(def Bob   {:loves #{:clojure :scheme}       :hates #{:c :c++ :algol}})
(def Ted   {:loves #{:clojure :lisp :scheme} :hates #{:algol :basic :c
                                                      :c++ :fortran}})

(merge-with clojure.set/intersection Alice Bob Ted)
;; -> {:loves #{:scheme :clojure}, :hates #{:c :c++}}
```

也可以创建一个递归合并函数，合并嵌套的映射表。

```
(defn deep-merge
  [& maps]
  (apply merge-with deep-merge maps))
(deep-merge {:foo {:bar {:baz 1}}}
            {:foo {:bar {:qux 42}}})
;; -> {:foo {:bar {:qux 42, :baz 1}}}
```

在前面的例子中可以看到，merge-with 是一个多功能的工具，我们可以用 + 来加总相同键的值，用 clojure.set/intersection 来找出多个集中都有的值，用递归函数 deep-merge 来递归地合并嵌套的映射表。merge-with 确实是一个非常强大的函数。

## 参阅

- 2.18 节“设置映射表中的键”。
- 2.22 节“一个键保存多个值”。

# 2.24 值的比较与排序

作者：Luke VanderHart

## 问题

要根据某个比较函数来比较两个值，或通过比较集合中的所有元素，对集合排序。

## 解决方案

用 `clojure.core/compare` 函数来比较两个值。它们必须是相互可比较的。例如，`double` 可以和有理数比较，因为它们都是数值，但字符串不能与向量比较。

如果第一个参数小于第二个参数，`compare` 就返回负数，等于就返回 0，大于就返回正数：

```
(compare 5 2)
;; -> 1

(compare 0.5 1)
;; -> -1

(compare (/ 1 4) 0.25)
;; -> 0

(compare "brewer" "aardvark")
;; -> 1
```

要比较整个集合，就将它传递给 `clojure.core/sort` 函数。`sort` 根据需要来应用 `compare`，返回排好序的序列。

例如，下面的代码将一个字符串分解为字符序列（`sort` 对它的参数调用 `seq`），然后对它们排序。结果重新连接成字符串，这样可读性较好：

```
(apply str (sort "The quick brown fox jumped over the lazy dog"))
;; -> "        Tabcddeeeefghhijklmnoooopqrrtuuvwxyz"
```

正如前面看到的，Clojure 的许多数据类型都有自然比较次序，这就是 `compare` 所用的次序。例如，数字、日期、字符串都像预期那样排序，从低到高，基于大家都理解和接受的固有次序。

如果需要排序的类型没有自然次序，或需要覆写自然次序（比如从高到低排序），也可以不用内建的比较函数。`sort` 允许指定一个定制的比较函数，执行期望的操作，来决定两个元素之间的相对次序。这个函数必须接受两个参数。它可以像 `compare` 那样返回值（也就是说，正整数、负整数或 0），或者返回布尔值（也就是一个谓词函数）。当且仅当第一个参数应该排在第二个参数前面时，这个谓词函数才应该返回 `true`。

这意味着可以将普通的 Clojure 谓词传递给 `sort`：

```
(sort > [1 4 3 2])
;; -> (4 3 2 1)

(sort < [1 4 3 2])
;; -> (1 2 3 4)
```

或者，也可以编写自己的比较函数。例如，下面例子中，定制的比较函数只关心字符串的长度，不关心其内容。如果字符串具有同样多的字符，它们就被判为相等，不论字符是什么：

```
(sort #(< (count %1) (count %2)) ["z" "yy" "zzz" "a" "bb" "ccc"])
;; -> ("z" "a" "yy" "bb" "zzz" "ccc")
```

# 讨论

在背后，Clojure 使用了 Java 内建的排序机制。Java 使用了稍加修改的归并排序算法，在绝大多数情况下效率都很高。它在最坏的情况下需要 $n \log(n)$ 次比较，如果输入已经相当有序，则性能接近 $O(n)$。

排序也是稳定的，这意味着如果比较函数判断两个元素相等，它们的相对位置在排序后不会改变。

尽管可以使用任何谓词作为比较函数，或编写自己的比较函数来返回正整数、负整数或 0，但实际的函数必须行为正确，才能有效。具体来说，它必须：

- *对于所有被排序的元素具有一致的全序*

  如果 `x` 排在 `y` 前面，`y` 排在 `z` 前面，那么 `x` 必须总是排在 `z` 前面。换句话说，对于给定的集合和比较函数，必须总是有唯一的完全确定的次序，排序函数不会导致任何冲突或不一致。

- *与.equals函数及Clojure的=函数一致*

  如果两个元素在逻辑上相等，那么就必须反映在比较函数中。如果使用整数返回值，函数就应该返回 `0`。如果使用谓词函数，如果 `x` 和 `y` 相等，`(pred x y)` 和 `(pred y x)` 应该返回同样的值。

- *没有副作用*

  在排序过程中，比较函数可能被调用任意次。

### 比较函数与JVM

Clojure 完全参与了 Java 的比较与排序机制。有自然次序的所有 Clojure 对象，都实现了 `java.lang.Comparable`（http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html）接口，实现了 `compareTo` 方法。

更重要的是，每个 Clojure 函数确实实现了 `java.util.Comprator`（http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html）接口。这意味着，可以将 Clojure 函数传递给任何需要 `java.util.Comparator` 的 Java 方法，它将用两个参数来调用该函数。这种机制允许将任意 Clojure 函数作为比较方法传递给 `sort`。函数对象本身实际上被用作 Java 的比较器，在 Clojure 函数上调用 Java 的 `.compare` 方法实际会调用该函数，要比较的两个值将作为参数传递给它。

因为谓词函数（返回 Boolean 值的函数）不能准确对应预期来自 `java.util.Comparator` 的返回值，即正整数、负整数和 0，所以 Clojure 自己负责它们之间的逻辑映射。如果用作比较方法的函数（即 `(pred x y)`）返回 `true`，实现将返回 `-1`，表明在给定的次序中，`x` 小于 `y`。如果不是，它会将参数反过来，再调用该函数。如果 `(pred x y)` 和 `(pred y x)` 都是 `false`，它就认为这两个对象是相等的，实现将返回 `0`。否则，它认为 `x` 大于 `y`，返回 `1`。

`sort-by`

有时候，希望集合排序时不是根据值本身，而是根据值的导出函数。例如，假定有如下数据，需要按名称的字母顺序排序。不幸的是，映射表没有自然顺序，所以需要告诉 Clojure，如何对数据排序：

```
(def people [{:name "Luke"   :role :author}
             {:name "Ryan"   :role :author}
             {:name "John"   :role :reviewer}
             {:name "Travis" :role :reviewer}
             {:name "Tom"    :role :reviewer}
             {:name "Meghan" :role :editor}])
```

一种选择是使用定制的比较方法，它提取 `:name` 键，然后调用 `compare`：

```
(sort #(compare (:name %1) (:name %2)) people)
;; -> ({:name "John", :role :reviewer}
;;     {:name "Luke", :role :author}
;;     {:name "Meghan", :role :editor}
;;     {:name "Ryan", :role :author}
;;     {:name "Tom", :role :reviewer}
;;     {:name "Travis", :role :reviewer})
```

但是，还有更容易的方法。`sort-by` 函数和 `sort` 的工作方式相同，但接受另一个函数 `keyfn` 作为参数，在对元素排序之前应用于元素。它不是按元素来排序，而是按对元素调用 `keyfn` 的结果来排序。

所以，将 `:name` 作为 `keyfn` 传入（正如 2.16 节中讨论的，关键字作为函数，在映射表中查找它们自己），可以调用：

```
(sort-by :name people)
;;-> ({:name "John", :role :reviewer}
;;     {:name "Luke", :role :author}
```

```
;;     {:name "Meghan", :role :editor}
;;     {:name "Ryan", :role :author}
;;     {:name "Tom", :role :reviewer}
;;     {:name "Travis", :role :reviewer})
```

像 sort 一样，sort-by 也接受一个可选的比较函数，用于比较 keyfn 提取出的值。

另一个例子，下面的表达式使用 str 函数作为 keyfn，对数字 1 至 20 进行排序，不是按照它们的数值，而是按照作为字符串的字典顺序（意味着“2”大于“10”，等等）。它也展示了利用定制的比较方法来指定结果为降序：

```
;; 降序字典序
(sort-by str #(* -1 (compare %1 %2)) (range 1 20))
;; -> (9 8 7 6 5 4 3 2 19 18 17 16 15 14 13 12 11 10 1)
```

### 数据结构的自然排序

某些复合数据结构，如果实现了 Comparable 接口，是同样的类型，并包含可比较的值，也可以进行比较。比较次序取决于实现。例如，默认情况下，向量的比较是先比较长度，然后对第一个值应用 compare，如果第一个值相等，再看第二个值，以此类推：

```
(sort [[2 1] [1] [1 2] [1 1 1] [2]])
;; -> ([1] [2] [1 2] [2 1] [1 1 1])
```

某些数据结构是不可比较的。例如，集按照定义是无序的，这意味着一般不可能得到有意义的大于、小于比较，所以没有提供比较方法。

## 参阅

- java.lang.Comparable 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html）。
- java.util.Comparator 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html）。
- 1.17 节“模糊比较”。
- 1.29 节“比较日期”。

# 2.25　从集合中移除重复元素

作者：John Cromartie

## 问题

有一个元素序列，需要移除所有重复元素，同时尽可能保持元素的次序。

## 解决方案

如果面对的元素序列是有界的，大小合适，就用 set 强制将集合转换成散列集，只包含不同的值：

```
(set [:a :a :g :a :b :g])
;; -> #{:a :b :g}
```

如果序列是无限的，或者需要保持次序，就用 distinct 返回一个惰性序列，它包含原集合中不同的值，保持出现的次序：

```
(distinct [:a :a :g :a :b :g])
;; -> (:a :g :b)
```

## 讨论

这两种方法之间有一些折中考虑。例如，set 利用了整个序列来得到新的集。因为这一点，set 不能用于过滤无限序列。distinct 与此不同，它是为惰性序列设计的。distinct 的值是另一个序列的惰性视图或投影，元素第一次出现时就产生新值：

```
(defn rand-int-seq
  "Returns an infinite sequence of ints from [0, n)"
  [n]
  (repeatedly #(rand-int n)))

;; 对无限序列应用 set* 永远不会 * 返回：
;; (set (rand-int-seq 10)) ; 不要这样做!

;; 但是，如果限制了序列，set 就有用
(set (take 10 (rand-int-seq 10)))
;; -> #{0 1 2 3 4 7 8 9}

;; distinct 任何情况都能用
(take 10 (distinct (rand-int-seq 10)))
;; -> (8 3 4 6 0 5 9 7 1 2)
```

因为 distinct 在看到新值时产生新值，所以它确实保持了原来的次序。set 不是这样，它返回无序的集。

如果 distinct 又有序，又惰性，又适用于任何长度，那 set 有什么好处？速度。使用 distinct 是最慢的选择，简单调用 set 会快两倍。

## 参阅

- 2.11 节“创建集”。

# 2.26 检测集合是否包含几个值中的一个

作者：John Touron

## 问题

需要确定集合是否包含几个值中的一个。

## 解决方案

使用 `some`，带一个集：

```
(some #{1 2} (range 10))
;; -> 1

(some #{10} (range 10))
;; -> nil
```

## 讨论

因为集可以用作函数，所以它们可以用作谓词，测试参数是不是集的元素。这种习惯用法将测试集合中的每个元素，返回第一个匹配的值，如果没有匹配，就返回 `nil`。但是，如果 `nil` 或 `false` 就是测试集的成员，就会有问题。考虑下面的例子：

```
(if (some #{nil} [1 2 nil 3])
  ::found
  ::not-found)
;; -> :user/not-found

(if (some #{false} [1 2 false 3])
  ::found
  ::not-found)
;; -> :user/not-found
```

因为 `some` 函数返回谓词函数的返回值，而不是 `true` 或 `false`，把它和包含 `nil` 或 `false` 的集一起使用，可能不是你的本意。如果集合中确实有 `nil` 或 `false`，它会返回该元素。最简单的方法就是单独测试 `nil` 和 `false`，利用 Clojure 自带的 `nil?` 和 `false?` 谓词函数：

```
(if (some nil? [nil false])
  ::found
  ::not-found)
;; -> :user/found

(if (some false? [nil false])
  ::found
  ::not-found)
;; -> :user/found
```

或者同时测试：

```
(if (some #(or (false? %)
               (nil? %))
      [nil false])
  ::found
  ::not-found)
;; -> :user/found
```

## 参阅

- 2.13 节“测试集成员”。
- 2.16 节“从映射表中取得值”。

# 2.27　实现定制的数据结构：红黑树（第一部分）

作者：Leonardo Borges

## 问题

需要在 Clojure 中实现一个有明确的性能特点的数据结构。

例如，需要对大型的、随机的、不断变化的数据集进行快速高效的内存查找。

## 解决方案

确定 Clojure 的核心数据结构不适合你面对的问题之后，第一步就是决定什么数据结构适合。

出于本实例的目的，假定需要选择并实现一种数据结构，能快速查找内存中大型的、随机的、不断变化的数据集。首先，二分查找树（BST）似乎是不错的解决方案。但它对排好序的数据集最有效。添加或移除大量的数据会让 BST 失衡，让它的性能退化到链表的水平。

红黑树（RBTs）和 BSTs 类似，但是能自动平衡。这就是适合该问题的数据结构。

下一步就是实现这个数据结构。RBT 的实现依赖于模式匹配。使用 `core.match`（https://github.com/clojure/core.match）来简化 RBT 的实现。请将 `[org.clojure/core.match "0.2.0"]` 加入项目依赖关系中，或用 `lein-try` 开始 REPL：

```
$ lein try org.clojure/core.match
```

首先，实现 RBT 的核心，即 `balance` 和 `insert-val` 函数。利用 `core.match`，可以根据树的形状，简洁地表示所需的行为：

```
(require '[clojure.core.match :refer [match]])

(defn balance
  "Ensures the given subtree stays balanced by rearranging black nodes
  that have at least one red child and one red grandchild"
  [tree]
  (match [tree]
         [(:or ;; 左子树红且左孙子树红
               [:black [:red [:red a x b] y c] z d]
               ;; 左子树红且右孙子树红
               [:black [:red a x [:red b y c]] z d]
               ;; 右子树红且左孙子树红
               [:black a x [:red [:red b y c] z d]]
               ;; 右子树红且右孙子树红
               [:black a x [:red b y [:red c z d]]])] [:red [:black a x b]
                                                             y
                                                             [:black c z d]]
               :else tree))

(defn insert-val
  "Inserts x in tree.
  Returns a node with x and no children if tree is nil.

  Returned tree is balanced. See also `balance`"
  [tree x]
  (let [ins (fn ins [tree]
              (match tree
                     nil [:red nil x nil]
                     [color a y b] (cond
                                     (< x y) (balance [color (ins a) y b])
                                     (> x y) (balance [color a y (ins b)])
                                     :else tree)))
        [_ a y b] (ins tree)]
    [:black a y b]))
```

有了插入和平衡，只剩下 `find-val` 函数要实现了，它检查值是否在 RBT 中。最容易的方法就是用 `match` 来分解单个树结点，并递归地寻找期望的值：

```
(defn find-val
  "Finds value x in tree"
  [tree x]
  (match tree
         nil nil
         [_ a y b] (cond
                     (< x y) (recur a x)
                     (> x y) (recur b x)
                     :else x)))
```

一切就续之后，就可以创建并查询 RBT 了：

```
(def rb-tree (reduce insert-val nil (range 4)))

rb-tree
```

```
;; -> [:black [:black nil 0 nil] 1 [:black nil 2 [:red nil 3 nil]]]

(find-val rb-tree 2)
;; -> 2

(find-val rb-tree 100)
;; -> nil
```

# 讨论

对于实现过红黑树，或者上过计算机科学课程、学过算法的人来说，balance 的实现似乎极其简单。原因有以下三点。

- 我们的红黑树是稳定持续的：对它的操作，诸如插入和平衡，不会破坏它。
- Balance 和 find-val 利用了 core.match，将逻辑编写为模式匹配。
- 节点表示为向量。

你很快会看到，后两点是相关的。

core.match 让我们能匹配数据结构的形状和值，同时执行结构绑定，这非常方便。例如，下面试图用两个子句来匹配 a-vector：

```
(def a-vector [1 2 3])

(match a-vector
       [_ y] (str "Got y: " y)
       [_ _ z] (str "Got z: " z))
;; -> "Got z: 3"
```

第一个子句匹配两个元素的向量，第二个子句匹配三个元素的向量。由于 a-vector 正好有三个元素，它匹配了第二个子句。在紧接着的表达式中，命名的值（如 z）绑定到它们匹配的位置。

这就是用向量来表示节点很方便的原因，对它们进行模式匹配毫不费力：

```
(def rb-node [:red nil 3 [:black nil 4 nil]])

(match rb-node
       [:red left value right]   (str "Red node with value: " value)
       [:black left value right] (str "Black node with value: " value))
;; -> "Red node with value: 3"
```

假定这个新的定制数据结构能满足性能标准，接下来做什么？（你打算对所有定制的数据结构跑个基准测试，对吗？）与内建的数据结构不同，这个定制的数据结构不支持 map 和 filter 这样的核心函数。

在本实例的第二部分，即 2.28 节，我们将改变这种状况，加入核心序列抽象。

## 参阅

- 本实例的第二部分，2.28 节“实现定制的数据结构：红黑树（第二部分）”，我们为 RBT 添加了序列功能。
- 维基百科上的红黑树（http://en.wikipedia.org/wiki/Red%E2%80%93black_tree）以更传统的方式探讨了这种有趣的数据结构。
- 对于本实例中的函数式方法，Chris Okasaki 的著作 *Purely Functional Data Structures*（http://www.amazon.com/Purely-Functional-Structures-Chris-Okasaki/dp/0521663504，Cambridge University Press）提供了很好的说明。该书探讨如何在函数式环境中高效地实现数据结构。作者选择使用 ML 和 Haskell，但概念也适合 Clojure，就像前面描述的那样。

# 2.28 实现定制的数据结构：红黑树（第二部分）

作者：Ryan Neufeld，最初由 Leonardo Borges 提交

## 问题

需要对定制的数据结构使用 Clojure 的核心序列函数（`conj`、`map`、`filter` 等）。

## 解决方案

本实例的第一部分实现了创建高效的红黑树所需的全部函数。缺失的部分是参与 Clojure 的序列抽象。

要参与序列抽象，最重要的一点是能够以序列的方式提供数据结构的值。内建的 `tree-seq` 很适合这个任务。但还需要一个步骤，因为 `tree-seq` 返回节点的序列，而不是值的序列。

下面是最终的 `rb-tree->seq` 函数：

```
(defn- rb-tree->tree-seq
  "Return a seq of all nodes in an red-black tree."
  [rb-tree]
  (tree-seq sequential? (fn [[_ left _ right]]
                          (remove nil? [left right]))
            rb-tree))

(defn rb-tree->seq
  "Convert a red-black tree to a seq of its values."
  [rb-tree]
  (map (fn [[_ _ val _]] val) (rb-tree->tree-seq rb-tree)))

(rb-tree->seq (-> nil
                  (insert-val 5)
                  (insert-val 2)))
;; -> (5 2)
```

既然 RBT 最像集，那就应该适应 PersistentSet 接口。扩展 IPersistentSet 和 IFn 协议，成为新的 RedBlackTree 类型，实现所有必要的函数。为 RedBlackTree 实现多重方法 print-method 也是个好主意，因为对于 RedBlackTree 来说，默认实现会失效：

```
(deftype RedBlackTree [tree]
  clojure.lang.IPersistentSet
  (cons [self v] (RedBlackTree. (insert-val tree v)))
  (empty [self] (RedBlackTree. nil))
  (equiv [self o] (if (instance? RedBlackTree o)
                    (= tree (.tree o))
                    false))
  (seq [this] (if tree (rb-tree->seq tree)))
  (get [this n] (find-val tree n))
  (contains [this n] (boolean (get this n)))
  ;; (disjoin [this n] ...) ;; 因为复杂而省略
  clojure.lang.IFn
  (invoke [this n] (get this n))
  Object
  (toString [this] (pr-str this)))
(defmethod print-method RedBlackTree [o ^java.io.Writer w]
  (.write w (str "#rbt " (pr-str (.tree o)))))
```

disjoin 和对应的 remove-val 函数留给读者作为练习。

现在可以像其他集合一样使用 RedBlackTree 实例了，特别是，它的实例就像集一样。

```
(into (->RedBlackTree nil) (range 2))
;; -> #rbt [:black nil 0 [:red nil 1 nil]]

(def to-ten (into (->RedBlackTree nil) (range 10)))

(seq to-ten)
;; -> (3 1 0 2 5 4 7 6 8 9)

(get to-ten 9)
;; -> 9

(contains? to-ten 9)
;; -> true

(to-ten 9)
;; -> 9

(map inc to-ten)
;; -> (4 2 1 3 6 5 8 7 9 10)
```

## 讨论

最后，加入序列抽象没有太多工作量。通过实现少数几个接口函数，2.27 节中的红黑树实现就能参与一系列面向序列的函数：map、filter、reduce……只要你说得出来。

本质上，clojure.lang.IPersistentSet 是一种抽象，代表数学上的集，这很符合树这种数据结构。但集不是列表或序列。那为什么说 RedBlackTree 参与序列抽象呢？

在 Clojure 中，扩展 clojure.lang.ISeq 接口的类型是真正的*序列*，表示由头和尾组成的逻辑列表。虽然 IPersistentSet 不是继承自 ISeq，但它们确实有共同的祖先。这两个接口都扩展了 clojure.lang.IPersistentCollection 及其父接口 clojure.lang.Seqable。幸运的是[1]，序列函数依赖于集合的 Seqable 接口，而不是 ISeq 接口。既然 RedBlackTree 可以被当成序列读入，它就是 Seqable 的，能被你熟悉和喜爱的所有序列函数操作。

IPersistentSet 接口中的大多数函数都是无需解释的，但有一些值得进一步解释。函数 cons 是一个历史名称，它通过向原有的列表添加一个值而得到新列表。seq 的作用是从集合得到序列，如果是空集就返回 nil：

IPersistentSet.java:

```
public interface IPersistentSet extends IPersistentCollection, Counted {
  public IPersistentSet disjoin(Object key);
  public boolean contains(Object key);
  public Object get(Object key);
}
```

IPersistentCollection.java:

```
public interface IPersistentCollection extends Seqable {
  int count();
  IPersistentCollection cons(Object o);
  IPersistentCollection empty();
  boolean equiv(Object o);
}
```

Seqable.java:

```
public interface Seqable {
  ISeq seq();
}
```

在所有 Seqable 的实现中，最有挑战的部分实际上是从底层的数据结构中得到一个序列。如果需要编写自己的惰性树遍历算法，难度就更大了。但幸运的是 Clojure 有内建的函数 tree-seq，就是用来完成这个任务的。利用 tree-seq 来生成节点的序列，编写 rb-tree->seq 转换函数很简单，它实现了 RedBlackTree 的惰性遍历，在遍历时得到节点的值。

注 1：实际上，要感谢设计者。

tree-seq 接受三个参数。

`branch?`

这是一个条件，在节点是枝（不是叶节点）时返回 true。对于 RedBlackTree 来说，sequential? 检查就足够了，因为每个节点都是一个向量。

`children`

这是一个函数，返回给定节点的所有子节点。

`root`

遍历开始的节点。

> tree-seq 执行深度优先遍历。根据红黑树的表示方式，这不是排序的遍历。

有了序列转换函数，很容易编写 seq 函数。类似地，编写 cons 和 empty 也不费事，只要利用已有的树函数。但相等测试可能有点难。

简单起见，我们选择只在 RedBlackTree 实例之间实现相等判断（equiv）。而且，该实现比较它们元素的排序序列。在这种情况下，equiv 回答的问题是“这些树拥有同样的值吗？”而不是“它们是同样的树吗？”这种区别很重要，在实现自己的数据结构时需要仔细考虑。

2.26 节讨论到，集有一大好处，它能像其他函数一样调用。为 RedBlackTree 提供这样的能力也很容易。通过实现 clojure.lang.IFn 接口的单参数函数 invoke，RedBlackTree 就能像其他函数一样调用（或者说，像集一样调用）：

```
(some (rbt [2 3 5 7]) [6])
;; -> nil

((rbt (range 10)) 3)
;; -> 3
```

即使完全实现了 IPersistentSet 接口，RedBlackTree 还是有一些不方便。例如，需要使用非专门设计的 / → Red  BlackTree 或 RedBlackTree. 函数来创建新的 RedBlackTree，然后再手工添加值。按照传统，许多内建的集合都提供便捷的方法来填充它们（更不必说像 [] 或 {} 这样的字面值标签了）。

很容易让 RedBlackTree 模仿 vec 和 vector：

```
(defn rbt
```

```
  "Create a new RedBlackTree with the contents of coll."
  [coll]
  (into (->RedBlackTree nil) coll))

(defn red-black-tree
  "Creates a new RedBlackTree containing the args."
  [& args]
  (rbt args))

(rbt (range 3))
;; -> #rbt [:black [:black nil 0 nil] 1 [:black nil 2 nil]]

(red-black-tree 7 42)
;; -> #rbt [:black nil 7 [:red nil 42 nil]]
```

你也许会注意到，打印不是序列抽象考虑的事，虽然对于开发用户友好和机器友好的数据结构来说，这肯定是考虑的重点。在 Clojure 中有两种类型的打印：`toString` 和基于 `pr` 的打印。`toString` 函数是为了在 REPL 中打印人可读的值，而 `pr` 系列的函数或多或少是为了打印 Clojure 读取程序可读的值。

要提供自己可读的 RBT 表示形式，必须为 `RedBlackTree` 实现 `print-method`（`pr` 的核心）。通过写成"带标签的字面值"格式（如 `#rbt`），就可以配置读取程序，将写下的值吸收并合成为一等对象：

```
(require '[clojure.edn :as edn])

;; Recall ...
(defmethod print-method RedBlackTree [o ^java.io.Writer w]
  (.write w (str "#rbt " (pr-str (.tree o)))))

(def rbt-string (pr-str (rbt [1 4 2])))
rbt-string
;; -> "#rbt [:black [:black nil 1 nil] 2 [:black nil 4 nil]]"

(edn/read-string rbt-string)
;; -> RuntimeException No reader function for tag rbt ...

(edn/read-string {:readers {'rbt ->RedBlackTree}}
                 rbt-string)
;; -> #rbt [:black [:black nil 1 nil] 2 [:black nil 4 nil]]
```

## 参阅

- 本实例的第一部分（2.27 节"实现定制的数据结构：红黑树（第一部分）"），其中定义了最初的红黑树实现。
- 4.14 节"读写 Clojure 数据"，以及 4.17 节"读取 Clojure 数据时处理未知的带标签字面值"，其中探讨了有关读取 Clojure 数据的更多内容。

第3章

# 广义计算

## 3.0 简介

商业中有句名言：没有哪个组织机构运行在理想环境中。这也适用于 Clojure。虽然 Clojure 提供了各种高效的工具和技术，但还是有一些活动和技术，出于某些原因，没能在软件中提供。有人称之为学术特质，或偶然的复杂性，但我们更愿意称之为生活的现实。

本章探讨了一些 Clojure 开发的主题，它们不太适合独立成章。比如说：

- 如何利用 Clojure 的开发生态系统？
- 抽象概念（例如多态）如何应用于 Clojure ？
- 什么是逻辑编程，何时会用到它？

## 3.1 运行最小的Clojure REPL

作者：John Cromartie

### 问题

在不装额外的工具的情况下，运行 Clojure REPL。

### 解决方案

从 http://clojure.org/downloads 下载并解压发行版，得到 Clojure 的 Java 包（JAR）。利用终

端，进入到解压 JAR 的目录，然后开始 Clojure REPL：

```
$ java -cp "clojure-1.5.1.jar" clojure.main
```

现在运行的就是交互式 Clojure REPL（“读取、求值、打印”循环）。输入表达式并回车，就能得到表达式的值。按 Ctrl-D 退出。

## 讨论

事实上，JVM 上的 Clojure 封装在一个 JAR 文件中，这样做有很多的好处。一个好处就是，这意味着 Clojure 从未真正安装。它只是一种依赖关系，像其他所有的 Java 库一样。替换掉一个文件，就可以容易地更换 Clojure 的版本。

我们分析一下这里的 `java` 调用。首先，设置了 Java 的类路径，包含 Clojure（在这个例子中，只包含 Clojure）：

```
-cp "clojure-1.5.1.jar"
```

全面解释类路径超出了本实例的讨论范围，只要知道它是一个位置列表，Java 将在这些位置寻找并加载类。JVM 类路径的全面讨论可以参考 http://docs.oracle.com/javase/tutorial/essential/environment/paths.html。在命令行的最后部分，我们指定了 Java 应该加载的类，并执行其 `main` 方法：

```
clojure.main
```

是的，`clojure.main` 确实是一个 Java 类。它看起来不像是典型的 Java 调用，这是因为 Clojure 的命名空间，它们会被编译成类，但不像 Java 类那样，习惯使用大写开头的名字。

这绝对是最小的 Clojure 环境，也是在安装了 Java 的系统上，运行 Clojure 所需的全部内容。当然，对于日常使用和开发，几乎肯定需要功能更丰富的解决方案，比如 Leiningen。

但在某些情况下，手工调整的 Java 调用可能是在环境中集成 Clojure 的最佳方式。在服务器上部署一个 JAR 文件很容易，不用安装更复杂的包，这一点尤其重要。

## 参阅

- Leiningen 网站（http://leiningen.org/）。
- 3.6 节“从命令行运行程序”。

# 3.2　交互式文档

作者：John Cromartie

## 问题

在 REPL 时，希望阅读函数文档。

## 解决方案

在 REPL 中，用 doc 宏打印函数文档：

```
user=> (doc conj)
 -------------------------
clojure.core/conj
([coll x] [coll x & xs])
  conj[oin]. Returns a new collection with the xs
    'added'. (conj nil item) returns (item). The 'addition' may
    happen at different 'places' depending on the concrete type.
```

在 REPL 中，用 source 宏打印函数源代码：

```
user=> (source reverse)
(defn reverse
  "Returns a seq of the items in coll in reverse order. Not lazy."
  {:added "1.0"
   :static true}
  [coll]
    (reduce1 conj () coll))
```

用 find-doc 找到文档匹配给定正则表达式的函数：

```
user=> (find-doc #"defmacro")
 -------------------------
clojure.core/definline
([name & decl])
Macro
  Experimental - like defmacro, except defines a named function whose
  body is the expansion, calls to which may be expanded inline as if
  it were a macro. Cannot be used with variadic (&) args.
 -------------------------
clojure.core/defmacro
([name doc-string? attr-map? [params*] body]
 [name doc-string? attr-map? ([params*] body) + attr-map?])
Macro
  Like defn, but the resulting function name is declared as a
  macro and will be used as a macro by the compiler when it is
  called.
```

## 讨论

Clojure 支持在线函数文档（稍后详述），以及其他元数据，让开发者可以在任何时候查看文档等内容。doc 和 source 宏只是 REPL 中方便的函数。

几乎在任何时候都可以查看 Clojure 的所有内部机制。如果你没有这样做过，下面的例子可能让你感到震撼：

```
user=> (source source)
(defmacro source
  "Prints the source code for the given symbol, if it can find it.
  This requires that the symbol resolve to a Var defined in a
  namespace for which the .clj is in the classpath.

  Example: (source filter)"
  [n]
  `(println (or (source-fn '~n) (str "Source not found"))))
```

如果知道 `source` 定义在 `clojure.repl` 命名空间中，通过对 `(source clojure.repl/source-fn)` 求值，就可以看到它到底如何取得源代码。

在大多数 REPL 实现中，像 `source` 和 `doc` 这样的 `clojure.repl` 宏只是引用到 `user` 命名空间。这意味着只要切换到另一个命名空间，未加限定的 `clojure.repl` 宏就不能访问了。可以为宏提供命名空间来绕过这个问题（用 `clojure.repl/doc`，而不是 `doc`），或者通过 `use` 那个命名空间，进行扩展应用：

```
user=> (ns foo)
foo=> (doc +)
CompilerException java.lang.RuntimeException: Unable to resolve symbol: doc
in this context, compiling:(NO_SOURCE_PATH:1:1)

foo=> (use 'clojure.repl)
nil

foo=> (doc +)
 -------------------------
clojure.core/+
([] [x] [x y] [x y & more])
  Returns the sum of nums. (+) returns 0. Does not auto-promote
  longs, will throw on overflow. See also: +'
```

用这种方式探索 Clojure，是学习核心函数和高级 Clojure 编程技术的极好方法。`clojure.core` 命名空间充满了唾手可得的高质量、高性能的代码。

## 参阅

- `clojure.repl` 的 API 文档（http://clojure.github.io/clojure/clojure.repl-api.html）。
- 3.3 节“探索命名空间”。

# 3.3　探索命名空间

作者：John Cromartie

## 问题

希望知道加载了哪些命名空间，它们有哪些公有的 var 可以访问。

## 解决方案

用 `loaded-libs` 获取当前加载的命名空间集。例如，在 REPL 中：

```
user=> (pprint (loaded-libs))
#{clojure.core.protocols clojure.instant clojure.java.browse
  clojure.java.io clojure.java.javadoc clojure.java.shell clojure.main
  clojure.pprint clojure.repl clojure.string clojure.uuid clojure.walk}
```

用 `dir` 打印命名空间中的公有 var：

```
user=> (dir clojure.instant)
parse-timestamp
read-instant-calendar
read-instant-date
read-instant-timestamp
validated
```

用 `ns-publics` 获取命名空间中符号到公有 var 的映射：

```
(ns-publics 'clojure.instant)
;; -> {read-instant-calendar #'clojure.instant/read-instant-calendar,
;;     read-instant-timestamp #'clojure.instant/read-instant-timestamp,
;;     validated #'clojure.instant/validated,
;;     read-instant-date #'clojure.instant/read-instant-date,
;;     parse-timestamp #'clojure.instant/parse-timestamp}
```

## 讨论

Clojure 中的命名空间是符号到 var 的动态映射。命名空间是不可访问的，除非它被引入。例如，在开始 REPL 或在 `ns` 声明中作为依赖关系时。只有在运行时，才知道可用的 Clojure 库和命名空间，这和典型的 Java 开发不同（关于包的几乎所有信息在编译时就知道了）。

动态本质的缺点是，必须知道要加载哪些命名空间，才能探索它们。

## 参阅

- `clojure.repl` API 文档（http://clojure.github.io/clojure/clojure.repl-api.html）。
- 3.2 节“交互式文档”。

# 3.4 尝试库而不指明依赖关系

作者：Mark Whelan

## 问题

希望在 REPL 中尝试一个库，而不必修改项目的依赖关系或创建新项目。

## 解决方案

用 Ryan Neufeld 的 `lein-try` 来发起 REPL。库依赖关系将自动满足。

要获得这种能力，首先要确保使用 Leiningen 2.1.3 或更新的版本。然后编辑 ~/.lein/profiles.clj 文件，将 `[lein-try "0.4.1"]` 添加到 `:user` 特性描述的 `:plugins` 向量中：

```
{:user {:plugins [[lein-try "0.4.1"]]}}
```

接下来就可以用你选择的库，体验几乎马上到来的满足感了：

```
$ lein try clj-time
Retrieving clj-time/clj-time/0.6.0/clj-time-0.6.0.pom from clojars
Retrieving clj-time/clj-time/0.6.0/clj-time-0.6.0.jar from clojars
nREPL server started on port 58981 on host 127.0.0.1
REPL-y 0.2.1
Clojure 1.5.1
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)
 Javadoc: (javadoc java-object-or-class-here)
    Exit: Control+D or (exit) or (quit)

user=>
```

## 讨论

注意，没必要给例子中的库指定版本号。`lein-try` 将自动抓取最新发布的版本。

当然，如果愿意，也可以指定库的版本。在库名后添加版本号就可以了：

```
$ lein try clj-time 0.5.1
#...
user=>
```

要快速了解用法选项，就用 **`lein help try`**：

```
$ lein help try
Launch REPL with specified dependencies available.
```

```
  Usage:

    lein try [io.rkn/conformity "0.2.1"] [com.datomic/datomic-free "0.8.4020.26"]
    lein try io.rkn/conformity 0.2.1
    lein try io.rkn/conformity # This uses the most recent version

  NOTE: lein-try does not require []

Arguments: ([& args])
```

作为一个 Clojure 工具，`lein-try` 提供了一种让任务更易完成的优雅的方法。根据你的喜好，用它从网上下载那些功能强大的库，不需要设置它们，瞬间就可以获得新的能力，享受魔法般的满足感。

## 参阅

- Leiningen 插件的官方例表（https://github.com/technomancy/leiningen/wiki/Plugins）。
- 前言中的“我们的金童 `lein-try`”。

# 3.5　运行Clojure程序

作者：John Cromartie

## 问题

希望通过 Clojure 源代码，运行有单个入口点的程序。

## 解决方案

要运行全是 Clojure 表达式的文件，只需将它的文件名作为参数传递给 `clojure.main`。

> 为了继续这个实例，可以从 http://clojure.org/downloads 下载 clojure.jar。

例如，若文件 my_clojure_program.clj 中包含内容：

```
(println "Hi.")
```

调用 `java` 命令，以 my_clojure_program.clj 作为最后的参数：

```
$ java -cp clojure.jar clojure.main my_clojure_program.clj
Hi.
```

在更结构化的项目中，可能有一些文件组织在 src/ 目录下。例如，给定文件 src/com/example/my_program.clj：

```
(ns com.example.my-program)

(defn -main [& args]
  (println "Hey!"))
```

要加载并运行 `-main` 函数，就用 `-m`/`--main` 选项指定期望的命名空间，并把 src 加入到类路径中（通过 `-cp`）：

```
$ java -cp clojure.jar:src clojure.main --main com.example.my-program
Hey!
```

## 讨论

尽管你会花许多的时间在 REPL 中对 Clojure 代码求值，但有时候也需要能够运行一个简单的“脚本”，其中都是 Clojure 表达式，或者运行带有 `-main` 入口点的、更结构化的 Clojure 应用。

不管哪种情况，都可以访问在脚本名后面传入的其他命令行参数，或是主命名空间的名称。

例如，假定有以下程序，放在名为 hello.clj 的文件中：

```
(defn greet
  [name]
  (str "Hello, " name "!"))

(doseq [name *command-line-args*]
  (println (greet name)))
```

直接调用这个 Clojure 程序，将得到预期的输出：

```
$ java -cp clojure.jar clojure.main hello.clj Alice Bob
Hello, Alice!
Hello, Bob!
```

这个简单的脚本有副作用，它在加载的时候打印输出。大多数 Clojure 代码不是以这种方式组织的。

因为通常总是希望将代码放在组织良好的命名空间中，所以可以用一个 `-main` 函数，为命名空间提供一个入口点。这样就能避开加载时的副作用，甚至可以在 REPL 中调整并调用 `-main` 函数，就像交互式开发时的所有其他函数一样。

假定将函数 `greet` 移至 `foo.util` 命名空间，项目的结构变成如下的样子：

```
./src/foo/util.clj
./src/foo.clj
```

foo 命名空间引入 foo.util 命名空间，并提供 -main 函数，像这样：

```
(ns foo
  (:require foo.util))

(defn -main
  [& args]
  (doseq [name args]
    (println (foo.util/greet name))))
```

如果调用 Clojure 时以 foo 作为“主”命名空间，它将调用 -main 函数，并带上提供的命令行参数：

```
$ java -cp clojure.jar:src clojure.main --main foo Alice Bob
Hello, Alice!
Hello, Bob!
```

你也注意到，-cp 选项后添加了 :src。这告诉 Java，执行时的类路径不仅包括 clojure.jar，也包括 src/ 目录下的内容。

## 参阅

- 3.6 节“从命令行运行程序”，学习如何从命令行运行 Leiningen 项目。
- 3.7 节“解析命令行参数”，学习如何在命令行应用中提供多个选项和标志。

# 3.6 从命令行运行程序

作者：Ryan Neufeld

## 问题

希望从命令行启动 Clojure 应用。

## 解决方案

对于所有 Leiningen 项目，用 **lein run** 命令从命令行启动应用。要继续这个实例，先创建一个新的 Leiningen 项目：

```
$ lein new my-cli
```

在项目的 project.clj 文件中，添加 :main 键，配置作为应用入口点的命名空间：

```
(defproject my-cli "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]]
  :main my-cli.core)
```

最后，为 project.clj 中配置的命名空间添加 `-main` 函数：

```
(ns my-cli.core)

(defn -main [& args]
  (println "My CLI received arguments:" args))
```

现在，用 **`lein run`** 来运行应用：

```
$ lein run
My CLI received arguments: nil

$ lein run 1 :foo "bar"
My CLI received arguments: (1 :foo bar)
```

# 讨论

事实证明，从命令行启动应用更容易。Leiningen 的 `run` 命令快速而方便地将应用与命令行联系起来，没有不必要的麻烦。在其基本形式中，无论项目的 project.clj 文件中将什么命名空间指定为 `:main`，`lein run` 都将调用它的 `-main` 函数。例如，设置 `:main my-cli.core` 都将会调用 `my-cli.core/-main`。或者，也可以不实现 `-main`，而在 `:main` 中指定带完整命名空间限定的函数（如 `my.cli.core/alt-main`），这个函数将替代 `-main` 被调用。

在前面的解决方案中，虽然打印出来的参数*看起来*像是 Clojure 数据，但它们实际上是普通字符串。对于简单的参数，你可能选择自己来解析这些字符串，或者，我们建议使用 `tools.cli` 库（https://github.com/clojure/tools.cli）。关于 `tools.cli` 的更多内容，参见 3.7 节。

尽管项目只能有一个默认的 `:main` 入口点，但也可以在命令行中设置 `-m` 选项，指向命名空间或函数，从而启动其他函数。如果将 `-m` 设置为命名空间（如 `my-cli.core`），该命名空间的 `-main` 函数就会被调用。如果将 `-m` 设置为函数的全名（如 `my-cli.core/alt-main`）则该函数将被调用。这个函数不需要带“-”前缀（表明它是一个 Java 方法），它只是必须接受可变数量的参数（就像 `-main` 通常那样）。

例如，可以在 `my.cli/core` 中添加 `add-main` 函数：

```
(ns my-cli.core)
```

```
(defn -main [& args]
  (println "My CLI received arguments:" args))

(defn add-main [& args]
  (->> (map #(Integer/parseInt %) args)
       (reduce + 0)
       (println "The sum is:")))
```

然后调用它，从命令行执行命令 **`lein run -m my-cli.core/add-main`**：

```
$ lein run -m my-cli.core/add-main 1 2 3
The sum is: 6
```

## 参阅

- 3.5 节“运行 Clojure 程序”，了解如何利用 **`java`** 运行普通的 Clojure 文件。
- 3.7 节“解析命令行参数”，了解如何利用 `tools.cli` 解析命令行参数。
- 8.2 节“将项目打包成 JAR 文件”，了解如何将应用打包成可执行的 JAR。
- 8.4 节“将应用作为守护进程运行”，了解如让应用成为守护进程。

# 3.7 解析命令行参数

作者：Ryan Neufeld，最初由 Nicolas Bessi 提交

## 问题

需要用 Clojure 编写命令行工具，解析输入参数。

## 解决方案

用 `tools.cli` 库（https://github.com/clojure/tools.cli）。

要继续这个实例，请将 `[org.clojure/tools.cli "0.2.4"]` 加入项目依赖关系中，或用 `lein-try` 开始 REPL：

```
$ lein try org.clojure/tools.cli
```

在项目的 `-main` 函数入口点中使用 `clojure.tools.cli/cli`，来解析命令行参数 [1]：

```
(require '[clojure.tools.cli :refer [cli]])

(defn -main [& args]
  (let [[opts args banner] (cli args
```

注 1：由于 `tools.cli` 如此方便实用，所以这个例子完全可以跑在 REPL 中。

```
                                   ["-h" "--help" "Print this help"
                                    :default false :flag true])]
      (when (:help opts)
        (println banner))))

;; 模拟在命令行中进入 -main
(-main "-h")
;; *out*
;; Usage:
;;
;;  Switches                 Default   Desc
;;  --------                 -------   ----
;;  -h, --no-help, --help    false     Print this help
```

## 讨论

Clojure 的 tools.cli 是一个简单的库，只有一个函数 cli，以及一个简洁的、面向对象的 API，用于指定参数应该如何解析。这个函数非常简便：提供参数向量和规格说明，得到解析后选项的映射表，可变长度的参数，以及帮助信息。它确实体现了良好的、可组合的函数式编程。

要配置选项如何解析，只需在 args 列表之后传入任意一个规格说明向量。例如，要指定 :port 参数，可以提供规格说明 ["-p" "--port"]。"-p" 不是必须的，而是为命令行选项提供一个字母的缩写方式（尤其是选项比较长时）。在返回的 opts 映射表中，选项名称的文本将转变成关键字（去掉 --）。例如，"--port" 将变成 :port，"--super-long-option" 将变成 :super-long-option。

一个有素养的命令行应用开发者会为每个选项加上描述。这将作为可选的字符串，跟在最终的参数名称后面：

```
["-p" "--port" "The incoming port the application will listen on."]
```

参数名称和描述之后的东西，都将被解释为键值对形式的选项。tools.cli 提供以下的选项：

:default
: 用户没有输入时的默认值。如果没有指定，:default 的默认值是 nil。

:flag
: 如果为真（不是 false 或 nil），表明参数就像一个标志或开关。该参数不会接受任何值作为其输入。

:parse-fn
: 用于解析参数值的函数。它可以用于将字符串转换成整数、浮点数，或其他数据类型。

:assoc-fn

用于将多个值组合成一个参数的函数。

下面是完整的例子：

```
(def app-specs [["-n" "--count" :default 5
                                :parse-fn #(Integer. %)
                                :assoc-fn max]
                ["-v" "--verbose" :flag true
                                  :default true]])

(first (apply cli ["-n" "2" "-n" "50"] app-specs))
;; -> {:count 50, :verbose true}

(first (apply cli ["--no-verbose"] app-specs))
;; -> {:count 5, :verbose false}
```

在写标志选项时，省略 :flag 选项，为参数名称加上 "[no-]" 前缀是一条可行的捷径。cli 会将这个参数规格说明解释为包含 :flag true，不用再指定：

```
["-v" "--[no-]verbose" :default true]
```

有一项功能 tools.cli 没有提供，就是钩入应用容器的启动生命周期。需要在 -main 函数中添加 cli 调用，并知道何时打印帮助信息。一般的用法是在 let 块中记录下 cli 的结果，并确定是否需要打印帮助信息。这对于确保参数的有效性也很有用（尤其是因为没有 :required 选项）：

```
(def required-opts #{:port})

(defn missing-required?
  "Returns true if opts is missing any of the required-opts"
  [opts]
  (not-every? opts required-opts))

(defn -main [& args]
  (let [[opts args banner] (cli args
                                ["-h" "--help" "Print this help"
                                  :default false :flag true]
                                ["-p" "--port" :parse-fn #(Integer. %)])]
    (when (or (:help opts)
              (missing-required? opts))
        (println banner))))
```

许多应用可能希望接受可变数量的参数，例如一组文件名。在大部分情况下，不需要做什么特殊工作，只要将它们放在其他选项后面。这些可变数量的参数将作为 cli 返回向量的第二个元素返回。

```
(second (apply cli ["-n" "5" "foo.txt" "bar.txt"] app-specs))
;; -> ["foo.txt" "bar.txt"]
```

但是，如果可变数量的参数看起来像标志，就需要另一个技巧。用 -- 作为一个参数，告诉 cli 之后的东西都是可变数量的参数。如果要调用另一个程序，它的参数先传给你的程序，这样做就很有用：

```
(second (apply cli ["-n" "5" "--port" "80"] app-specs))
;; -> Exception '--port' is not a valid argument ...

(second (apply cli ["-n" "5" "--" "--port" "80"] app-specs))
;; -> ["--port" "80"]
```

在 REPL 中完成了应用的选项解析尝试之后，你可能希望尝试通过 **lein run** 来调用选项。就像应用需要用 -- 来表明哪些参数是传递给后续程序，也必须用 -- 告诉 **lein run**，哪些参数是给你的程序的，哪些参数是给它的：

```
# 如果 app-specs 被配置给了一个项目……
$ lein run -- -n 5 --no-verbose
```

## 参阅

- 3.6 节“从命令行运行程序”，了解从命令行启动应用的更多内容。
- 4.1 节“写入 STDOUT 和 STDERR”，了解输入和输出流。
- 8.2 节“将项目打包成 JAR 文件”，了解如何将应用打包成可执行的 JAR 文件。
- 要构建 Ncurses 风格的应用，参见 clojure-lanterna（http://sjl.bitbucket.org/clojure-lanterna/），它封装了 Lanterna 终端输出库。

# 3.8 创建定制的项目模板

作者：Travis Vachon

## 问题

经常需要创建新的、类似的项目，希望有容易的方法来生成定制的样板文件。或者，你在开发一个开源项目，希望为用户提供一种使用你的软件的容易的方法。

## 解决方案

Leiningen 模板为 Clojure 程序员提供了一种容易的方式，用一条命令自动生成定制的项目样板文件。通过创建一个简单 Web 服务的模板，我们来探索一下。

首先，用 **lein new template cookbook-sample-template-<github_user>** 生成一个新模板。将 <github_user> 替换为你自己的 GitHub 用户名，因为将要把这个模板发布到 Clojars，它需要一个唯一的名字。在这个例子中，使用 clojure-cookbook 作为 GitHub 用户名：

```
$ lein new template cookbook-sample-template-clojure-cookbook
Generating fresh 'lein new' template project.

$ cd cookbook-sample-template-clojure-cookbook
```

在 src/leiningen/new/<project-name>/project.clj 中创建一个新的项目文件模板，内容如下：

```
(defproject {{ns-name}} "0.1.0"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"

            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]])
```

因为要创建 Web 服务的模板，希望 Clojure 的 `ring` 和 `ring-jetty-adapter` 也默认提供，所以将它们加到 `:dependencies` 中：

```
:dependencies [[org.clojure/clojure "1.5.1"]
               [ring "1.1.8"]
               [ring/ring-jetty-adapter "1.2.0"]]
```

接下来，打开模板定义（src/leiningen/new/<project-name>.clj），将 `project.clj` 添加到要生成的文件列表中。将 `sanitize-ns` 添加到命名空间的 `:require` 指令中，提供清洁的命名空间字符串：

```
(ns leiningen.new.cookbook-sample-template-clojure-cookbook
  (:require [leiningen.new.templates :refer [renderer
                                             name-to-path
                                             ->files
                                             sanitize-ns]]
            [leiningen.core.main :as main])) ; ❶

(def render (renderer "cookbook-sample-template-clojure-cookbook"))

(defn cookbook-sample-template-clojure-cookbook
  "FIXME: write documentation"
  [name]
  (let [data {:name name
              :ns-name (sanitize-ns name)                   ; ❷
              :sanitized (name-to-path name)}]
    (->files data
            ["project.clj" (render "project.clj" data)]       ; ❸
            ["src/{{sanitized}}/foo.clj" (render "foo.clj" data)])))
```

❶ 将 `sanitize-ns` 添加到 `:require` 指令中。

❷ 提供 `:ns-name` 作为简写形式 `name`。

❸ 将 project.clj 添加到模板的文件列表中。

好的模板为用户提供了一个基本框架，可以在此基础上构建。创建新的文件 src/leiningen/new/<project-name>/site.clj，提供 Web 服务逻辑的梗概：

```
(ns {{ns-name}}.site
    "My website! It will rock!"
    (:require [ring.adapter.jetty :refer [run-jetty]]))

(defn handler [request]
  {:status 200
   :headers {"Content-Type" "text/html"}
   :body "Hello World"})

(defn -main []
  (run-jetty handler {:port 3000}))
```

回到模板的 project.clj 文件，为 `:main` 选项添加键和值，表明 `my-website.site` 是项目的核心可执行命名空间：

```
:main {{ns-name}}.site
```

回到模板定义（<project-name>.clj），将两个 `foo.clj` 引用改成 `site.clj`。同时删除文件 src/leiningen/new/<project-name>/foo.clj。

```
;; ...
["src/{{sanitized}}/site.clj" (render "site.clj" data)])))
```

要在本地测试该模板，将模板项目的根路径作为当前路径，并运行：

```
$ lein install
$ lein new cookbook-sample-template-clojure-cookbook my-first-website --snapshot
$ cd my-first-website
$ lein run
# ... Leiningen noisily fetching dependencies ...
2013-08-22 16:41:43.337:INFO:oejs.Server:jetty-7.6.8.v20121106
2013-08-22 16:41:43.379:
  INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:3000
```

如果 `lein` 不能找到模板，并打印出错误，就要用 **`lein upgrade`**，确认使用的是最新版本。

要向其他用户提供该模板，就要将它发布到 Clojars。首先，打开模板项目的 project.clj，将版本改成发布版，因为默认 `lein` 仅使用非 SNAPSHOT 的模板：

```
(defproject cookbook-sample-template-clojure-cookbook/lein-template "0.1.0"
;; ...
```

接下来，访问 clojars.org，创建 Clojars 账户，然后从模板项目的根路径部署：

```
$ lein deploy clojars
```

其他用户现在可以用你的模板名称作为 **`lein new`** 的第一个参数，创建新的项目了。Leiningen 将自动从 Clojars 获取项目模板：

```
$ lein new cookbook-sample-template-clojure-cookbook my-second-website
```

## 讨论

Leiningen 利用 Clojars 作为知名模板来源。如果将模板名称传递。给 `lein new`，它先会在本地 Maven 库中按名称查找该模板。如果没找到，就会在 http://clojars.org 上查找相应命名的模板。如果找到，就会下载该模板，用它来创建新项目。结果得到几乎像魔法一样的项目创建接口，非常适合 Clojure 程序员快速接触新技术。

下载了项目模板之后，Leiningen 将利用 src/leiningen/new/<project-name>.clj 来创建新项目。这个文件可以彻底定制，以创建满足要求的复杂模板。我们将探讨这个文件，并讨论模板开发者可用的一些工具。

我们先声明一个符合模板名称的命名空间，并请求一些有用的函数，它们是 Leiningen 专为模板开发提供的 `leiningen.new.templates` 还包含了各种其他函数，在开发自己的模板之前值得了解一下，因为你在开发过程中遇到的问题，这个库可能已经解决了。在这个例子中，`name-to-path` 和 `sanitize-ns` 将帮助创建一些字符串，它们将在一些位置替换文件模板中的内容：

```
(ns leiningen.new.cookbook-sample-template-clojure-cookbook
  (:require [leiningen.new.templates :refer [renderer
                                             name-to-path
                                             ->files
                                             sanitize-ns]]))
```

新项目的生成是加载一组 mustache 模板文件，并按照一组命名的字符串对它们进行渲染。`renderer` 函数创建了一个函数，它在模板名称确定的位置，查找 mustache 模板。在这个例子中，它将在 src/leiningen/new/cookbook_sample_template_clojure_cookbook/ 中查找模板：

```
(def render (renderer "cookbook-sample-template-clojure-cookbook"))
```

延续配置的惯例，Leiningen 将在这个命名空间中查找一个与模板同名的函数。在这个函数中可以执行任何 Clojure 代码，这意味着项目的生成可以达到任意的复杂度：

```
(defn cookbook-sample-template-clojure-cookbook
  "FIXME: write documentation"
  [name]
```

渲染程序将使用这些数据，利用提供的模板创建新的项目文件。在这个例子中，我们让模板能知道项目名称、从该名称导出的命名空间，以及基于该名称的清洁的路径：

```
(let [data {:name name
            :ns-name (sanitize-ns name)
            :sanitized (name-to-path name)}]
```

最后，我们向 `->files`（读作“to files”）函数传入一个列表，包含文件名 / 内容的三元组。

文件名确定新项目中文件将放在哪里。内容是利用前面定义的 render 函数生成的。render 接受到模板文件的相对路径，以及我们创建的键值映射：

```
(->files data
        ["project.clj" (render "project.clj" data)]
        ["src/{{sanitized}}/site.clj" (render "site.clj" data)])))
```

Mustache 模板非常简单，只实现简单的键替换。例如，下面的代码片段用于为新项目的主文件 site.clj 生成 ns 语句：

```
(ns {{ns-name}}.site
    "My website! It will rock!"
    (:require [ring.adapter.jetty :refer [run-jetty]]))
```

Leiningen 模板是强大的工具，帮 Clojure 开发者省去了建立项目的辛苦工作。更重要的是，它们对开源开发者非常有价值，可以展示他们的项目，让潜在用户极其容易地开始接触一个不熟悉的软件。如果你开发 Clojure 已经有一段时间，哪怕只是刚刚开始，都值得花时间立刻尝试一下！

## 参阅

- Leiningen 模板文档（https://github.com/technomancy/leiningen/blob/master/doc/TEMPLATES.md）。
- leiningen.new.templates 命名空间的源代码（https://github.com/technomancy/leiningen/blob/master/src/leiningen/new/templates.clj）。
- mustache 模板（http://mustache.github.io/）。

# 3.9 构建具有多态行为的函数

作者：Ryan Neufeld，最初由 David McNeil 提交

## 问题

希望创建一些函数，其行为随传入的参数不同而不同。例如，希望开发一组灵活的几何函数。

## 解决方案

要实现运行时多态，最容易的方法是通过手工的、基于映射的分派，利用 cond 或 condp 这样的函数：

```
(defn area
  "Calculate the area of a shape"
  [shape]
```

```
  (condp = (:type shape)
    :triangle (* (:base shape) (:height shape) (/ 1 2))
    :rectangle (* (:length shape) (:width shape))))

(area {:type :triangle :base 2 :height 4})
;; -> 4N

(area {:type :rectangle :length 2 :width 4})
;; -> 8
```

但这种方法有点原始：area 与分派和多种形状的实现绑在一起，都在一个函数中。利用 defmulti 和 defmethod 宏来定义多重函数，将分离分派和实现，引入一定程度的可扩展性：

```
(defmulti area
  "Calculate the area of a shape"
  :type)

(defmethod area :rectangle [shape]
  (* (:length shape) (:width shape)))

(area {:type :rectangle :length 2 :width 4})
;; -> 8

;; 尝试取得新形状的面积……
(area {:type :circle :radius 1})
;; -> IllegalArgumentException No method in multimethod 'area' for
;;    dispatch value: :circle ...

(defmethod area :circle [shape]
  (* (. Math PI) (:radius shape) (:radius shape)))

(area {:type :circle :radius 1})
;; -> 3.141592653589793
```

这要好一些，但如果希望加入新的几何函数，如 perimeter，代码就开始散乱了。如果使用多重方法，就要为每个函数重复分派逻辑，因为组合爆炸而需要编写大量的实现。如果这些函数和实现能够分组并写在一起，就更好了。

利用 Clojure 的 protocol 机制来定义协议接口，并用具体实现对它进行扩展：

```
;; 定义 Shape 对象的 " 形状 "
(defprotocol Shape
  (area [s] "Calculate the area of a shape")
  (perimeter [s] "Calculate the perimeter of a shape"))

;; 定义具体的 Shape，Rectangle
(defrecord Rectangle [length width]
  Shape
  (area [this] (* length width))
  (perimeter [this] (+ (* 2 length)
                       (* 2 width))))
(->Rectangle 2 4)
```

```
;; -> #user.Rectangle{:length 2, :width 4}

(area (->Rectangle 2 4))
;; -> 8
```

## 讨论

正如这个实例所示，在 Clojure 中有许多不同方式来实现多态。虽然前面的例子最后用了协议作为实现多态的方法，但选择哪种技术，并没有硬性的规定和快捷的方法。每种方法都有自己特有的一些折中，在实现多态时需要考虑。

第一种方法是简单的基于映射的多态，使用 `condp`。回顾一下就会发现，它不是在 Clojure 中构建几何库的好选择，但这不是说它毫无用处。这种方法适合小规模编程（http://en.wikipedia.org/wiki/Programming_in_the_large_and_programming_in_the_small）：你可以在 REPL 中，用 `cond` 作为协议早期迭代的原型，或者用在不准备定义新类型的地方。

要注意的是，除了 `cond` 之外，还有别的技术来实现基于映射的分派。其中一种技术是用一个分派映射表，一般实现为键与函数的映射表。

下一种方法是采用多重方法。不像基于 `cond` 的多态，多重方法分离了分派和实现。由于这一点，它们可以在创建之后扩展。多重方法的定义使用了 `defmulti` 宏，它的行为类似于 `defn`，但指定了一个分派函数，而不是一种实现。

让我们来分解一个相当简单的多重方法的 `defmulti` 声明，即 `area` 函数：

```
(defmulti area ;                    ❶
  "Calculate the area of a shape" ; ❷
  :type) ;                          ❸
```

❶ 这个多重方法的函数名。

❷ 描述该函数的文档字符串。

❸ 分派函数。

使用关键字 `:type` 作为分派函数，并不能充分说明多重方法的灵活性：它们的能力强得多。多重方法允许对调用的参数执行任意复杂的检查。

如果选择像 `:type` 这样的映射表查询作为分派函数，就隐含了函数的元数（它接受的参数个数）。因为关键字作为接受一个参数（映射表）的函数，所以 `area` 是一个单“元数”函数。其他函数会隐含不同的元数。对于多重函数，常见的模式是使用匿名函数，让多重函数对元数的要求更明显：

```
(defmulti ingest-message
  "Ingest a message into an application"
  (fn [app message] ;      ❶
```

```
    (:priority message)) ; ❷
  :default :low) ;         ❸
```

❶ ingest-messages 接受两个参数，app 和 message。

❷ message 的不同处理取决于它的优先级。

❸ 如果 message 中没有 :priority 键，默认的优先级是 :low。

如果没有指定，默认的分派值是 :default。

```
(defmethod ingest-message :low [app message]
  (println (str "Ingesting message " message ", eventually...")))

(defmethod ingest-message :high [app message]
  (println (str "Ingesting message " message ", now.")))

(ingest-message {} {:type :stats :value [1 2 3]})
;; *out*
;; Ingesting message {:type :stats :value [1 2 3]}, eventually...

(ingest-message {} {:type :heartbeat :priority :high})
;; *out*
;; Ingesting message {:type :heartbeat, :priority :high}, now.
```

在目前为止的所有例子中，总是根据一个值分派。多重方法也支持所谓的*多重分派*，即一个函数可以根据多种因素派发。

通过返回一个向量，而不是一个值，可以作出更动态的决定：

```
(defmulti convert
  "Convert a thing from one type to another"
  (fn [request thing]
    [(:input-format request) (:output-format request)])) ; ❶

(require 'clojure.edn)
(defmethod convert [:edn-string :clojure] ;                ❷
  [_ str]
  (clojure.edn/read-string str))

(require 'clojure.data.json)
(defmethod convert [:clojure :json] ;                      ❸
  [_ thing]
  (clojure.data.json/write-str thing))

(convert {:input-format :edn-string
          :output-format :clojure}
         "{:foo :bar}")
;; -> {:foo :bar}

(convert {:input-format :clojure
          :output-format :json}
         {:foo [:bar :baz]})
;; -> "{\"foo\":[\"bar\",\"baz\"]}"
```

❶ convert 多重方法根据输入和输出格式来分派。

❷ convert 的一个实现将 edn 字符串转成 Clojure 数据。

❸ 类似地，一个实现将 Clojure 数据转成 JSON。

但这些能力都有代价。由于多重方法非常动态，所以可能比较慢。而且，没有很好的方法将几组相关的多重方法放到一个要么全有、要么全无的包中[2]。如果主要考虑速度或实现一个完整的接口，协议可能是更好的选择。

Clojure 的协议机制提供了可扩展的多态，以及类似于 Java 接口的快速分派。它与多重方法有一个值得注意的区别：协议只能执行单分派（基于类型）。

协议是通过 defprotocol 宏来定义的，它接受名称、可选的文档字符串，以及不定数目的命名方法签名。方法签名由几个部分组成：名称，至少一个类型签名，以及可选的文档字符串。类型签名的第一个参数总是该对象本身，因为 Clojure 按照这个参数的类型来分派。也许例子是阐明 defprotocol 语法最容易的方法：

```
(defprotocol Frobnozzle
  "Basic methods for any Frobnozzle"
  (blint [this x] "Blint the frobnozzle with x") ;                     ❶
  (crand [this f] [this f x] (str "Crand a frobnozzle with another " ; ❷
                                  "optionally incorporating x")))
```

❶ 函数 blint，带有一个额外参数 x。

❷ 多元数函数 crand，带有可选的 x 参数。

定义了协议之后，有许多种方式来实现它。deftype、defrecord 和 reify 都在创建对象的同时，定义了协议的实现。deftype 和 defrecord 创建了新的命名类型，而 reify 创建了匿名类型。每种形式都表明协议被扩展了，紧接着是协议的每个方法的具体实现：

```
;; deftype 的语法类似，但并不适用于不可变的形状
(defrecord Square [length]
  Shape ;                          ❶
  (area [this] (* length length)) ; ❷
  (perimeter [this] (* 4 length))
  ;                                ❸
  )

(perimeter (->Square 1))
;; -> 4

;; 计算平行四边形的面积，不定义记录
(area
  (let [b 2
        h 3]
```

注 2：这就是说，在将行为扩展到它自己的类型时，不能强制多重方法实现所有要求的方法。

```
      (reify Shape
        (area [this] (* b h))
        (perimeter [this] (* 2 (+ b h))))))
;; -> 6
```

❶ 表明要实现的协议。

❷ 实现它的所有方法。

❸ 对所有要实现的协议，重复第 1 步和第 2 步。

> **类型与记录的区别**
>
> 类型和记录的语法非常类似，所以很难理解何时该用哪一个。
>
> Chas Emerick 在 *Clojure Programming* (O'Reilly, http://www.clojurebook.com/）一书的附录中解释得非常好：
>
> > 类对领域值建模，因此受益于类似哈希映射表的功能和语义吗？用 `defrecord`。
> >
> > 需要定义可变的字段吗？用 `deftype`。
>
> 这下清楚了。

要在已有的类型上实现协议，就要用到内建的 `extend` 函数族（`extend`、`extend-type` 和 `extend-protocol`）。这些函数不是定义新类型，而是在已有的类型上定义实现。

## 参阅

- 多重方法与层级的官方文档（http://clojure.org/multimethods），深度探讨了多重方法。这份文档也介绍了层级，因为它也与多重方法有关，而本实例没有介绍。
- 协议的官方文档（http://clojure.org/protocols），深度探讨了协议，包括协议与接口的关系。
- 2.28 节“实现定制的数据结构：红黑树（第二部分）”，有实现协议的具体例子。
- 3.10 节“扩展内建的类型”，例如使用 `extend` 及其方便的宏 `extend-type` 和 `extend-protocol`。

# 3.10 扩展内建的类型

作者：David McNeil

## 问题

需要用自己的函数扩展内建的类型。

## 解决方案

假定需要在核心的 java.lang.String 类型上添加领域特定的函数。在这个例子中，要为 String 添加 first-name 和 last-name 函数。定义一个协议，包含需要的函数。协议声明了函数签名：

```
(defprotocol Person
  "Represents the name of a person."
  (first-name [person])
  (last-name [person]))
```

扩展 java.lang.String 类的类型：

```
(extend-type String
  Person
  (first-name [s]  (first  (clojure.string/split s #" ")))
  (last-name [s] (second (clojure.string/split s #" "))))
```

现在可以在字符串上调用你的函数：

```
(first-name "john")
;; -> "john"

(last-name "john smith")
;; -> "smith"
```

## 讨论

既然已经有了多重方法，为什么要用协议？一个理由就是速度：协议只按第一个参数的类型来分派。而且，协议允许对扩展进行分组和命名。这样推断哪些函数归属哪个类型就容易得多，并且确保了合适的、完整的实现。

只有当你是协议或类型的作者时，才将协议扩展为一个类型，这是一种很好的实践方式。这样能避免违反最初作者的假定。

如果已经有一些函数要使用，那么就使用 extend，而不是 extend-type：

```
(defn first-word [s]
  (first (clojure.string/split s #" ")))

(defn second-word [s]
  (second (clojure.string/split s #" ")))

(extend String
  Person
  {:first-name first-word
   :last-name second-word})
```

## 参阅

- Jörg W Mittag 在 StackOverflow 上回答了“Expression Problem”（http://stackoverflow.com/questions/4509782/simple-explanation-of-clojure-protocols/4513556#4513556），很好地解释了为什么协议会存在。

# 3.11 用 core.async 解除消费者和生产者的耦合

作者：Daemian Mack

## 问题

需要在消费者和生产者之间引入显式的队列，解除它们的耦合。

例如，如果要建立一个 Web 仪表板，取得 Twitter 消息，这个应用必须将这些消息持久到数据库，并通过服务器发送事件（SSE）发布给浏览器。

## 解决方案

在组件间引入显式的队列，让它们能够异步通信，使它们更容易独立管理，并释放计算资源。利用 `core.async` 库（https://github.com/clojure/core.async）来引入和协调异步信道。

要继续这个实例，请用 `lein-try` 开始 REPL：

```
$ lein try org.clojure/core.async
```

请考虑以下代码，它展示了一种同步的方式：

```
(defn database-consumer
  "Accept messages and persist them to a database."
  [msg]
  (println (format "database-consumer received message %s" msg)))

(defn sse-consumer
  "Accept messages and pass them to web browsers via SSE."
  [msg]
  (println (format "sse-consumer received message %s" msg)))

(defn messages
  "Fetch messages from Twitter."
  []
  (range 4))

(defn message-producer
  "Produce messages and deliver them to consumers."
  [& consumers]
  (doseq [msg (messages)
```

```
          consumer consumers]
    (consumer msg)))

(message-producer database-consumer sse-consumer)
;; *out*
;; database-consumer received message 0
;; sse-consumer received message 0
;; database-consumer received message 1
;; sse-consumer received message 1
;; database-consumer received message 2
;; sse-consumer received message 2
;; database-consumer received message 3
;; sse-consumer received message 3
```

接收到的每条消息直接传递给了 message-producer 的消费者。这种实现方式非常脆弱，只要有慢速的消费者，整个管道就会慢慢停下来。

要实现异步处理，就用 clojure.core.async/chan 引入显式的队列。将工作包装在 core.async 中的一种 clojure.core.async/go 形式中，实现工作的异步执行：

```
(require '[clojure.core.async :refer [chan sliding-buffer go
                                      go-loop timeout >! <!]])

(defn database-consumer
  "Accept messages and persist them to a database."
  []
  (let [in (chan (sliding-buffer 64))]
    (go-loop [data (<! in)]
             (when data
               (println (format "database-consumer received data %s" data))
               (recur (<! in))))
    in))

(defn sse-consumer
  "Accept messages and pass them to web browsers via SSE."
  []
  (let [in (chan (sliding-buffer 64))]
    (go-loop [data (<! in)]
             (when data
               (println (format "sse-consumer received data %s" data))
               (recur (<! in))))
    in))

(defn messages
  "Fetch messages from Twitter."
  []
  (range 4))

(defn producer
  "Produce messages and deliver them to consumers."
  [& channels]
  (go
   (doseq [msg (messages)
```

```
            out channels]
      (<! (timeout 100))
      (>! out msg))))

(producer (database-consumer) (sse-consumer))
;; *out*
;; database-consumer received data 0
;; sse-consumer received data 0
;; database-consumer received data 1
;; sse-consumer received data 1
;; database-consumer received data 2
;; sse-consumer received data 2
;; database-consumer received data 3
;; sse-consumer received data 3
```

## 讨论

> 在所有好的程序中，组件或子系统之间必须停止直接通信，这样的时刻终将到来。
>
> —— Rich Hickey
> Clojure core.async Channels

这段代码比最初的实现要长。它给我们带来了什么好处？

最初的实现是呆板的。它没有提供对消费者延迟的控制，因此非常容易拖延。利用信道来缓冲通信，异步地完成工作，我们就在生产者和消费者之间创建了服务边界，让它们能够尽可能独立地运行。

让我们仔细检查一个新的消费者，理解发生了什么变化。

现在消费者不是通过函数调用来收到消息，而是从带缓冲的信道取得消息。过去消费者（如 `database-consumer`）每次消费一条消息，现在使用了 `go-loop`，连续不断地从生产者那里消费消息。

在传统的回调式代码中，完成这样的事情需要在许多函数间切分逻辑，导致所谓的“回调地狱”。`core.async` 的一个好处就是，它让你编写直线式代码，更为直接明了：

```
(defn database-consumer
  "Accept messages and persist them to a database."
  []
  (let [in (chan (sliding-buffer 64))] ; ❶
    (go-loop [data (<! in)]            ; ❷
             (when data                ; ❸
               (println (format "database-consumer received data %s" data))
               (recur (<! in))))       ; ❹
    in))
```

❶ 这里给信道缓存大小是 64。`sliding-buffer` 变量表明，如果信道累积超过 64 个未读的

值，老的值就开始“丢失”，它更注重新的值，而不是历史的完整性。使用 dropping-buffer 的优化方向是相反的。

❷ go-loop 是 core.async 中等价于 while true 这样的循环。这个 go-loop 从输入信道（in）中取值（<!），作为其初始值。

❸ 因为信道在关闭时返回 nil，所以只要能从信道中读取数据，就表明还有工作要做。

❹ 要 recur 这个 go-loop 到开始位置，就从信道中取下一个值，以它为参数调用 recur。

因为 go-loop 块是异步的，取值调用（<!）会停在那里，直到有值投入信道。go-loop 块的剩余部分（这里是 println 调用）就挂起了。因为信道是作为 database-consumer 函数的值返回的，所以系统的其他部分（即生产者）可以在等待取值时，自由地写入信道。写入信道的第一个值将满足读取调用，让 go-loop 块的剩余部分继续下去。

这个消费者现在是异步的，读取值直到信道关闭。因为信道是带缓冲的，现在就对系统的弹性有了一定程度的控制。例如，缓冲区允许消费者比生产者延迟一定的时间。

让 producer 异步所需的改动更少：

```
(defn producer
  [& channels]
  (go
   (doseq [msg (messages)
           out channels]  ; ❶
     (<! (timeout 100))   ; ❷
     (>! out item))))     ; ❸
```

❶ 对每个消息和信道……

❷ 从 timeout 信道读取，模拟短暂停顿的效果……

❸ 用 >! 向信道写入消息。

尽管操作是异步的，它们仍是串行式执行的。使用无缓冲的消费者信道意味着，如果一个消费者从信道中读取的速度太慢，管道将停止，生产者将不能向信道写入新值。

## 参阅

- core.async 还有更高级的机制来分配和协调信道。更多信息，请参阅 core.async 概述（http://clojure.github.io/core.async/）。
- 5.8 节“并发使用 ZeroMQ”，了解 core.async 如何通过 ZeroMQ 通信。

# 3.12　用core.match为Clojure表达式制作解析器

作者：Chris Frisz

## 问题

需要解析 Clojure expressions，例如，从输入变为宏，变为不同的表示形式（如映射表）。

作为例子，请考虑一个大幅简化的 Clojure 版本，包含以下的表达式类型：

- 以合法的 Clojure 符号表示的变量；
- `fn` 表达式只接受一个参数，它的函数体也是一个合法的表达式；
- 将语言中的一个合法表达式作用于另一个合法表达式。

可以用下面的语法表示这种语言：

```
Expr = var
     | (fn [var] Expr)
     | (Expr Expr)
```

## 解决方案

使用 `core.match` 对输入进行模式匹配，然后将该表达式作为映射表的映射表返回。

在开始前，请将 `[org.clojure/core.match "0.2.0"]` 加入你的项目依赖关系中，或用 `lein-try` 开始 REPL：

```
$ lein try org.clojure/core.match
```

现在，利用 `clojure.core.match/match` 对该语言的语法进行编码：

```
(require '[clojure.core.match :refer (match)])

(defn simple-clojure-parser
  [expr]
  (match [expr]
    [(var :guard symbol?)] {:variable var}
    [(['fn [arg] body] :seq)] {:closure
                               {:arg arg
                                :body (simple-clojure-parser body)}}
    [([operator operand] :seq)] {:application
                                 {:operator (simple-clojure-parser operator)
                                  :operand (simple-clojure-parser operand)}}
    :else (throw (Exception. (str "invalid expression: " expr)))))

(simple-clojure-parser 'a)
;; -> {:variable a}

(simple-clojure-parser '(fn [x] x))
;; -> {:closure {:arg x, :body {:variable x}}}

(simple-clojure-parser '((fn [x] x) a))
;; -> {:application
;;     {:operator {:closure {:arg x, :body {:variable x}}}
```

```
;;      :operand {:variable a}}}

;; fn 表达式只能接受一个参数!
(simple-clojure-parser '(fn [x y] x))
;; -> Exception invalid expression: (fn [x y] x) ...
```

# 讨论

core.match 中的 match 语句由两个基本部分组成。第一部分是要匹配的 var 组成的向量。在这个例子中，就是 [expr]。这个向量不限于一个元素，它可以包含多个希望匹配的元素。第二部分是一个可变的“问题 / 答案对”列表。问题是一个向量，表示 var 向量必须具备的形状。像 cond 一样，“答案”是 var 满足问题时应该返回的东西。

问题在 core.match 中有几种形式。下面是对前面例子的解释。

- 第一个匹配模式是 [(var :guard symbol?)]，它匹配了语法中变量的情况，将匹配的表达式与 var 绑定。特殊的 :guard 形式将谓词 symbol? 应用于 var，只在 symbol? 返回 true 时，才返回答案。
- 第二个模式是 [(['fn [arg] body] :seq)]，匹配 fn 的情况[3]。请注意，特殊的 ([...] :seq) 语法是用于匹配列表的，在这里用来代表 fn 表达式。也要注意，要匹配字面值 fn，它必须在匹配模式中带上引号。有趣的是，因为这个解析器也应该接受 body 表达式，所以它进行了自递归调用 (simple-clojure-parser body)，在匹配模式的右边。
- 对第三个 :application 模式，解析器再次使用 ([...] :seq) 匹配一个列表。因为在 fn 表达式的函数体部分，operator 和 operand 表达式都应该被解析器接受，所以它对每一个进行了递归调用。

最后，如果给出的表达式不匹配这三种接受的模式，解析器将抛出异常。如果偶然向解析器提交了不合法的表达式，这多少提供了点有帮助的信息。

这样编写解析器得到的是简明的代码，非常像它所针对的输入。或者，也可以用条件表达式（if 或 cond）来写，显式地解构输入。为了说明代码长度和清晰度的区别，请看这个函数，它只解析了该语言的 fn 表达式：

```
(defn parse-fn
  [expr]
  (if (and (list? expr)
           (= (count expr) 3)
           (= (nth expr 0) 'fn)
           (vector? (nth expr 1))
           (= (count (nth expr 1)) 1))
    {:closure {:arg (nth (nth expr 1) 0)
               :body (simple-clojure-parser (nth expr 2))}}
```

注 3：针对 fn 的匹配模式可以（也应该）包含对 arg 的检查，确保它是一个符号，但这里为了简洁省去了。

```
            (throw (Exception. (str "unexpected non-fn expression: " expr)))))
```

看到这个版本需要多少代码，才能表示 fn 表达式的同样属性吗？没用 match 的版本不仅需要更多的代码，而且 if 检测也不如 match 模式那样像表达式的结构。而且，match 将匹配的输入自动绑定到变量名上，不用手工通过 let 来绑定，或重复编写同样的列表访问代码(就像上面的 parse-fn 中用的 (nth expr) 那样)。显然，match 版要更容易阅读和维护。

## 参阅

- core.match 的维基概述页面（https://github.com/clojure/core.match/wiki/Overview），更全面地介绍了这个库的功能。

# 3.13 用core.logic查询层级图

作者：Ryan Senior

## 问题

有一个像图一样的层级数据结构，序列化成节点的展平列表，需要进行查询。例如，有一个电影元数据的图，表示为一些“实体 – 属性 – 值”的三元组。用标准的 seq 函数来编写这段代码相当繁琐，也容易出错。

## 解决方案

core.logic 库是领域特定语言（DSL）miniKanren 的 Clojure 实现，目的是进行逻辑编程。它的描述式风格很适合查询展平的层级数据。

要继续这个实例，就用 lein-try 开始 REPL：

```
$ lein try org.clojure/core.logic
```

首先需要待查询的数据集。例如，已经将电影元数据的图表示为三元组的列表：

```
(def movie-graph
  [;; "Newmarket Films" 工作室
   [:a1 :type :FilmStudio]
   [:a1 :name "Newmarket Films"]
   [:a1 :filmsCollection :a2]

   ;; 由 Newmarket Films 制作的电影集
   [:a2 :type :FilmCollection]
   [:a2 :film :a3]
   [:a2 :film :a6]
```

```
;; 电影 "Memento"
[:a3 :type :Film]
[:a3 :name "Memento"]
[:a3 :cast :a4]

;; 电影与演职人员的关联（演员、导演、制作人等）
[:a4 :type :FilmCast]
[:a4 :director :a5]

;; "Memento" 的导演
[:a5 :type :Person]
[:a5 :name "Christopher Nolan"]

;; 电影 "The Usual Suspects"
[:a6 :type :Film]
[:a6 :filmName "The Usual Suspects"]
[:a6 :cast :a7]

;; 电影与演职人员的关联（演员、导演、制作人等）
[:a7 :type :FilmCast]
[:a7 :director :a8]

;; "The Usual Suspects" 的导演
[:a8 :type :Person]
[:a8 :name "Bryan Singer"]])
```

有了这些数据后，如何查询呢？如果采用仓促的模型，就会费力地一个一个“连接这些节点”，使用过滤器、映射表和条件[4]。但利用 core.logic，就可以用描述式的逻辑语句，来连接这些节点。

例如，要回答问题“哪些导演在给定的工作室执导过电影？”就利用 clojure.core.logic/fresh 创建一些节点（*逻辑变量*），并用 clojure.core.logic/membero 连接它们。最后，调用 clojure.core.logic/run* 来得到所有可能的解决方案：

```
(require '[clojure.core.logic :as cl])

(defn directors-at
  "Find all of the directors that have directed at a given studio"
  [graph studio-name]
  (cl/run* [director-name]
    (cl/fresh [studio film-coll film cast director]
      ;; 将最初的工作室名称连接到电影集
      (cl/membero [studio :name studio-name] graph)
      (cl/membero [studio :type :FilmStudio] graph)
      (cl/membero [studio :filmsCollection film-coll] graph)

      ;; 将所有电影集连接到它们的每部电影
      (cl/membero [film-coll :type :FilmCollection] graph)
      (cl/membero [film-coll :film film] graph)
```

注 4：我的天哪！

```
    ;; 然后是电影到演职人员
    (cl/membero [film :type :Film] graph)
    (cl/membero [film :cast cast] graph)

    ;; 演职人员到 type :director
    (cl/membero [cast :type :FilmCast] graph)
    (cl/membero [cast :director director] graph)

    ;; 最后，连接到导演姓名
    (cl/membero [director :type :Person] graph)
    (cl/membero [director :name director-name] graph))))

(directors-at movie-graph "Newmarket Films")
;; -> ("Christopher Nolan" "Bryan Singer")
```

# 讨论

miniKanren 是用 Scheme 写的领域特定语言，目的是把逻辑编程语言（如 Prolog）中的许多优点引入到 Scheme 中。David Nolen 创建了 miniKanren 的 Clojure 实现，关注点是性能。逻辑编程语言的一个好处就是描述式风格。利用 `core.logic`，我们可以说在图中要找什么，而不用说 `core.logic` 应该到哪里去找。

一般来说，所有 `core.logic` 查询都始于库提供的 `run` 宏之一，`clojure.core.logic/run` 返回有限数目的解决方案，`clojure.core.logic/run*` 返回所有的解决方案。

`run` 宏的第一个参数是“目标”，用于存储查询结果的变量。在前面的解决方案中，就是 `director-name` 变量。接下来是 `core.logic` 程序的主体。程序由一些“逻辑变量”构成（用 `clojure.core.logic/fresh` 创建），连接到值或受逻辑语句的限制。

`run` 透露出一个信息，我们的编程范式将转变为逻辑编程。在 `core.logic` 程序中，使用“合一”（unification），而不是传统的变量赋值和顺序表达式求值。合一通过用值来替换变量，尝试让两个表达式在句法上等价。`core.logic` 程序中的语句能以任何次序出现。例如，可以用 `clojure.core.logic/==` 使 `1` 和 `q` 合一。

```
(cl/run 1 [q]
  (cl/== 1 q))
;; -> (1)

(cl/run 1 [q]
  (cl/== q 1))
;; -> (1)
```

`core.logic` 也能够让列表和向量的内容合一，找到正确的替换，让两个表达式等价：

```
(cl/run 1 [q]
  (cl/== [1 2 3]
         [1 2 q]))
```

```
;; -> (3)

(cl/run 1 [q]
  (cl/== ["foo" "bar" "baz"]
         [q     "bar" "baz"]))
;; -> ("foo")
```

从技术上说，合一是一种关系，将第一种形式与第二种形式关联起来。这是 core.logic 要解决的一种迷题。在前面的例子中，q 是一个逻辑变量，core.logic 的职责是将一个值与 q 绑定，让合一（clojure.core.logic/== 关系）的左边和右边在句法上是等价的。如果没有绑定能满足这个迷题，解决方案就不存在：

```
;; 无法让一个值既是 1 又是 2
(cl/run 1 [q]
  (cl/== 1 q)
  (cl/== 2 q))
;; -> ()
```

fresh 是创建更多逻辑变量的一种方法：

```
(cl/run 1 [q]
  (cl/fresh [x y z]
    (cl/== x 1)
    (cl/== y 2)

    (cl/== z 3)
    (cl/== q [x y z])))
;; -> ([1 2 3])
```

正如 clojure.core.logic/== 是两种形式之间的关系，clojure.core.logic/membero 是列表中的元素与列表之间的关系：

```
(cl/run 1 [q]
  (cl/membero q [1]))
;; -> (1)

(cl/run 1 [q]
  (cl/membero 1 q))
;; -> ((1 . _0))
```

第一个例子是问列表 [1] 中的所有成员，刚好只有 1。第二个例子正好相反，问成员包含 1 的列表。圆点记号表示不合适的尾部，其中有 _0。这意味着 1 可以自己出现在列表中，或者后面跟着一串其他数字、字符串、列表等。_0 是一个未绑定的变量，因为除了 1 是成员外，这个列表没有进一步的约束。

> clojure.core.logic/run* 是一个宏，请求所有可能的解决方案。请求所有包含 1 的列表将无法结束。

合一也能用 clojure.core.logic/membero，看到结构内部：

```
(cl/run 1 [q]
  (cl/membero [1 q 3] [[1 2 3] [4 5 6] [7 8 9]]))
;; -> (2)
```

逻辑变量的生存期是整个程序执行过程，确保在多个语句中使用同一个逻辑变量：

```
(let [seq-a [["foo" 1 2] ["bar" 3 4] ["baz" 5 6]]
      seq-b [["foo" 9 8] ["bar" 7 6] ["baz" 5 4]]]
  (cl/run 1 [q]
    (cl/fresh [first-item middle-item last-a last-b]
      (cl/membero [first-item middle-item last-a] seq-a)
      (cl/membero [first-item middle-item last-b] seq-b)
      (cl/== q [last-a last-b]))))
;; -> ([6 4])
```

前面的例子没有指定 first-item，只是它对 seq-a 和 seq-b 来说应该是一样的。core.logic 用提供的数据将值绑定到满足约束的变量。middle-item 也是一样。

在这个基础上，我们可以遍历解决方案中描述的图：

```
(cl/run 1 [director-name]
  (cl/fresh [studio film-coll film cast director]
    (cl/membero [studio :name "Newmarket Films"] graph)
    (cl/membero [studio :type :FilmStudio] graph)
    (cl/membero [studio :filmsCollection film-coll] graph)

    (cl/membero [film-coll :type :FilmCollection] graph)
    (cl/membero [film-coll :film film] graph)

    (cl/membero [film :type :Film] graph)
    (cl/membero [film :cast cast] graph)

    (cl/membero [cast :type :FilmCast] graph)
    (cl/membero [cast :director director] graph)

    (cl/membero [director :type :Person] graph)
    (cl/membero [director :name director-name] graph)))
;; -> ("Christopher Nolan")
```

上面的代码和最初的解决方案之间有一点小差异：没有用 clojure.core.logic/run* 请求所有解决方案，而是用了 clojure.core.logic/run 1。对于查询 Newmarket Films 的导演，程序有多个答案。请求多个答案就会返回更多结果，不需要改动其他代码。

对前面的查询稍作改动，就会明显改变结果。将 "Newmarket Films" 换成一个新的、未绑定的变量，就会返回所有工作室的所有导演。如果愿意，也可以创建宏，来减少一些代码重复。

关系式解决方案对这个问题有一个好处，就是能从一些值生成图。

```
(first
  (cl/run 1 [graph]
    (cl/fresh [studio film-coll film cast director]
      (cl/membero [studio :name "Newmarket Films"] graph)
      (cl/membero [studio :type :FilmStudio] graph)
      (cl/membero [studio :filmsCollection film-coll] graph)

      (cl/membero [film-coll :type :FilmCollection] graph)
      (cl/membero [film-coll :film film] graph)

      (cl/membero [film :type :Film] graph)
      (cl/membero [film :cast cast] graph)

      (cl/membero [cast :type :FilmCast] graph)
      (cl/membero [cast :director director] graph)

      (cl/membero [director :type :Person] graph)
      (cl/membero [director :name "Baz"] graph))))
;; -> ([_0 :name "Newmarket Films"]
;;     [_0 :type :FilmStudio]
;;     [_0 :filmsCollection _1]
;;     ...)
```

对于小型的图，`membero` 足够快。大型的图会有性能问题，因为 `core.logic` 将多次遍历列表，找到这些元素。利用 `clojure.core.logic/to-stream` 和一些基本索引，能大幅提高查询性能。

## 参阅

- 由Daniel P. Friedman、William E. Byrd和Oleg Kiselyov合著的The Reasoned Schemer（MIT Press）。
- `core.logic` 的维基页面（https://github.com/clojure/core.logic/wiki）。
- miniKanren 的网站（http://minikanren.org/）。
- `core.logic` 的代码库（https://github.com/clojure/core.logic），包含了使用 `clojure.core.logic/to-stream` 的例子。
- `core.match`（https://github.com/clojure/core.match）（非合一）的匹配库，有一些类似的原理，3.12 节“用 `core.match` 为 Clojure 表达式制作解析器”中有简单描述。

# 3.14 演奏儿歌

作者：Chris Ford

# 问题

希望编码演奏儿歌，启发孩子开始编程。

# 解决方案

用 Overtone（https://github.com/overtone/overtone）来演奏歌曲。

在开始前，请将 `[overtone "0.8.1"]` 加入你的项目依赖关系中，或用 `lein-try` 开始 REPL[5]：

```
$ lein try overtone
```

开始先定义一首老儿歌的旋律：

```
(require '[overtone.live :as overtone])

(defn note [timing pitch] {:time timing :pitch pitch})

(def melody
  (let [pitches
         [0 0 0 1 2
          ; Row, row, row your boat,
          2 1 2 3 4
          ; Gently down the stream,
          7 7 7 4 4 4 2 2 2 0 0 0
          ; (take 4 (repeat "merrily"))
          4 3 2 1 0]
          ; Life is but a dream!
        durations
         [1 1 2/3 1/3 1
          2/3 1/3 2/3 1/3 2
          1/3 1/3 1/3 1/3 1/3 1/3 1/3 1/3 1/3 1/3 1/3 1/3
          2/3 1/3 2/3 1/3 2]
        times (reductions + 0 durations)]
      (map note times pitches)))

melody
;; -> ({:time 0, :pitch 0}   ;Row,
;;     {:time 1, :pitch 0}   ;row,
;;     {:time 2, :pitch 0}   ;row
;;     {:time 8/3, :pitch 1} ;your
;;     {:time 3N, :pitch 2}  ;boat
;;     ...)
```

将这段旋律转换成具体的调，即用代表该调的函数转换每个音符的音高：

```
(defn where [k f notes] (map #(update-in % [k] f) notes))

(defn scale [intervals] (fn [degree] (apply + (take degree intervals))))
(def major (scale [2 2 1 2 2 2 1]))
```

注 5：如果在 Linux 上运行 Overtone，有一些另外的安装考虑。更详细的安装指导，请参见 Overtone 的维基页面（https://github.com/overtone/overtone/wiki#installation）。

```
(defn from [n] (partial + n))
(def A (from 69))

(->> melody
(where :pitch (comp A major)))
;; -> ({:time 0, :pitch 69} ; Row,
;;     {:time 1, :pitch 69} ; row,
;;     ...)
```

将这段旋律转换成具体的拍子，即用代表拍子的函数转换每个音符的音长：

```
(defn bpm [beats] (fn [beat] (/ (* beat 60 1000) beats)))

(->> melody
  (where :time (comp (from (overtone/now)) (bpm 90))))
;; -> ({:time 1383316072169, :pitch 0}
;;     {:time 4149948218507/3, :pitch 0}
;;     ...)
```

现在，定义一种乐器，用它来演奏这段旋律。下面例子中的合成乐器是一个简单的正弦波，它的振幅和音长是由一个包络来控制的：

```
(require '[overtone.live :refer [definst line sin-osc FREE midi->hz at]])

(definst beep [freq 440]
  (let [envelope (line 1 0 0.5 :action FREE)]
    (* envelope (sin-osc freq))))

(defn play [notes]
  (doseq [{ms :time midi :pitch} notes]
    (at ms (beep (midi->hz midi)))))

;; 确保扬声器打开……
(->> melody
  (where :pitch (comp A major))
  (where :time (comp (from (overtone/now)) (bpm 90)))
play)
;; -> <music playing on your speakers>
```

如果儿歌是一种轮唱（round），像“Row, Row, Row Your Boat”这样，可以用它来自我伴奏：

```
(defn round [beats notes]
  (concat notes (->> notes (where :time (from beats)))))

(->> melody
  (round 4)
  (where :pitch (comp A major))
  (where :time (comp (from (overtone/now)) (bpm 90)))
  play)
```

# 讨论

音符是具有一定音高、延续一段时间的声音。歌曲是一系列音符。因此在 Clojure 中，我

们可以简单地将音乐表示为时间 / 音高对的序列。这种表示在结构上与西方音乐记谱法非常类似，五线谱上的圆点表示一段时间，音高由它的水平和垂直位置来决定。但不像传统的音乐记谱法，Clojure 的表示可以由函数式编程技术来操纵。

西方音乐片段，如“Row, Row, Row Your Boat”，不是由任意音高组成的。在一段旋律中，音符通常被限定在所有可能音高的一个子集中，称为*音阶*。

这里采用的方法，是将音高表示为整数，表示它们出现在音阶上的位置，称为*度*。所以，`0` 度表示音阶中的第 `1` 个音高，`4` 度表示音阶中的第 `5` 个音高。

这简化了旋律的描述，因为我们不用担心指定的音高会跑到音阶外面去。这也让我们能够选择不同的音阶，而不用重写旋律。

要使用度，就需要一个函数，将度翻译成真正的音高。因为“Row, Row, Row Your Boat”是在大调音阶，所以需要一个函数来表示该音阶。我们发现，在主音阶中，相邻两个音高之间距离两个位置或一个位置（音乐家称之为全音或半音）。我们定义了名为 major（大调）的函数，它接收度，输出它所表示的半音。

我们的音高仍然不太对，因为它们是相对于这个片段的最低音符的。需要建立一个音乐参考点，用于解释我们的度。A 调常被乐队用作参考点，所以我们用它作为音乐上的 0 度。换言之，我们将用 A 大调来演奏“Row, Row, Row Your Boat”，现在 0 度就表示 A。

注意，可以简单地结合针对大调和针对 A 的函数，得到组合的 A 大调函数。

对时间也需要类似的转换。每个音调的时间用*拍子*（beat）来表示，但我们需要用毫秒来表示。我们可以用当前的系统时间作为临时参考点，意味着这个片段将从现在开始（而不是从 Unix 纪元的起始时间开始）。“Row, Row, Row Your Boat”是一种轮唱，这意味着如果作为它自己的伴唱，离开一定的拍子，也是和谐的。作为额外的修饰，我们提供了第二个版本的旋律，比第一个版本慢 4 拍。

我们建议你尝试进行一些调整，可以改变速度，或使用不同的调（提示一下，小调的全音和半音模式是 `[2 1 2 2 1 2 2]`）

我们也建议你思考，如何用这种方式来对一系列事件建模，应用于其他领域。将时间系列表示为一个序列，然后对整个序列进行转换，用这种思路来描述问题是简单的、灵活的，也是可组合的方式。

音乐是美妙动人的。它也很适合在函数式编程语言中建模。希望你的孩子也同意这一点。

## 参阅

- Clojure 中的音乐环境 Overtone（https://github.com/overtone/overtone）。

第 4 章

# 本地I/O

## 4.0 简介

在前面几章中，我们完成了大量的工作，但是显而易见，它们必须在某个地方实际派上用场。如何将数据输入 Clojure 程序？更重要的是，如何输出？本章即探讨了如何在本地计算机上实现输入和输出，因为，这就是大多数应用程序的数据能派上用场的地方。

与本地计算机通信的方式和媒介有很多种。我们与什么通信？以什么方式？采用什么格式？这有点像经典的棋盘游戏“妙探寻凶”（Clue）：它是控制台上作为命令行参数的纯文本？还是文件中作为配置信息的 Clojure 数据？本章将探讨文件、格式以及 GUI 和控制台风格的应用等等。

虽然不可能列出所有的组合，但我们希望本章能给你留下深刻的印象，让你了解各种可能。Clojure 中大多数好的解决方案可以组合使用，这很方便。你可以毫不费力地应用本章中的各种实例来满足你的需求。

## 4.1 写入STDOUT和STDERR

作者：Alan Busby

### 问题

希望写入 `STDOUT` 和 `STDERR`。

## 解决方案

默认情况下，print 和 println 函数将传递给它们的内容输出到 STDOUT：

```
(println "This text will be printed to STDOUT.")
;; *out*
;; 这段文字将输出到 STDOUT。

(do
  (print "a")
  (print "b"))
;; *out*
;; ab
```

将 *out* 绑定改为 *err*，以便输出到 STDERR，而不是 STDOUT：

```
(binding [*out* *err*]
  (println "Blew up!"))
;; *err*
;; Blew up!\n
```

## 讨论

在 Clojure 中，*out* 和 *err* 是动态绑定的 var，分别对应于应用环境内建的 STDOUT 和 STDERR 流。

Clojure 的所有打印函数，如 print 和 println，都用 *out* 绑定作为输出目的地。因此，可以将它重新绑定到 *err*（利用 binding），从而将打印信息的目的地从 STDOUT 改为 STDERR。其他打印函数包括 pr、prn、printf，以及另外几种。

*out* 绑定的值不限于操作系统的流，*out* 可以是任何类似流的对象。这让打印函数变得非常强大。它们可以输出到文件、套接字，或任何其他管道。内建的函数 clojure.java.io/writer 是多功能的输出流构造器：

```
;; 创建一个 writer，指向文件 foo.txt，并输出到该文件
(def foo-file (clojure.java.io/writer "foo.txt"))
(binding [*out* foo-file]
  (println "Foo, bar."))

;; 没有什么输出到原来的 *out*

;; 当然，要关闭该文件
(.close foo-file)
```

## 参阅

- pr 的文档（http://clojure.github.io/clojure/clojure.core-api.html#clojure.core/pr）和源代码（https://github.com/clojure/clojure/blob/c6756a8bab137128c8119add29a25b0a88509900/src/clj/clojure/core.clj#L3325），以便更好地理解基于 *out* 输出的原理。

- `clojure.java.io/writer` 的文档（http://clojure.github.io/clojure/clojure.java.io-api.html#clojure.java.io/writer），了解创建 writer 对象的更多信息。

## 4.2 从控制台读入一次击键

作者：John Jacobsen

### 问题

通过 `stdin` 的控制台输入通常是按行缓冲的，但你可能会希望从控制台读入一次击键，不要缓冲。

### 解决方案

利用 JLine 库（https://github.com/jline/jline2）的 `ConsoleReader`，这个 Java 库负责处理控制台输入。

JLine 类似于 BSD 的 editline 和 GNU 的 readline。要继续这个实例，请用 **`lein new keystroke`** 创建一个新库。在 project.clj 中，将 `[jline "2.11"]` 添加到 `:dependencies` 向量中。

在文件 src/keystroke/core.clj 中，利用 `ConsoleReader`，从终端读取字符：

```
(ns keystroke.core
  (:import [jline.console ConsoleReader]))

(defn show-keystroke []
  (print "Enter a keystroke: ")
  (flush)
  (let [cr (ConsoleReader.)
        keyint (.readCharacter cr)]
    (println (format "Got %d ('%c')!" keyint (char keyint)))))
```

### 讨论

像大多数语言一样，Java 中的控制台 I/O 是有缓冲的，`flush` 将最初的命令写入标准输出流。但是，输入默认也是有缓冲的。JLine 库提供了 `ConsoleReader` 对象，它的 `readCharacter` 方法可以避免输入缓冲。但是，在 REPL 中测试 `show-keystroke` 时要谨慎：

```
$ lein repl
user=> (require '[keystroke.core :refer [show-keystroke]])
user=> (show-keystroke)
Enter a keystroke:
;; HANGS!
```

为了将控制台输入正确地连接到 REPL，请使用 **`lein trampoline repl`**（这里的 `<r>` 表示

用户输入了字母 r）：

```
$ lein trampoline repl
user=> (require '[keystroke.core :refer [show-keystroke]])
user=> (show-keystroke)
Enter a keystroke: <r>Got 114 ('r')!
nil
user=>
```

`lein trampoline` 是必需的，因为默认情况下，Leiningen 实际上是在另一个 JVM 进程中运行 REPL 及其相关的控制台 I/O，与你的代码不同。`trampoline` 选项强制 Leiningen 在同一个进程中运行 REPL 和你的代码，像"跳蹦床"一样来回切换控制。通常这是看不见的，但是如果执行的代码希望直接使用控制台，就会有问题。

如果在 REPL 之外执行你的程序（就像通常那样，运行 Clojure 写的命令行应用），这就不是问题。

## 参阅

- 如果想要更丰富的、基于终端的界面，像 C 的 curses 库提供的那样，那么 `clojure-lanterna`（http://sjl.bitbucket.org/clojure-lanterna/）库是个不错的起点。

# 4.3 执行系统命令

作者：Mark Whelan 和 Ryan Neufeld

## 问题

希望向底层的操作系统发出一个命令，并获得其输出。

## 解决方案

利用 `clj-commons-exec` 库，在本地操作系统上执行 shell 命令。

要继续本实例，请用 `lein-try` 开始 REPL：

```
$ lein try org.clojars.hozumi/clj-commons-exec "1.0.6"
```

调用 `clj-commons-exec/exec` 函数，带上要执行的命令，将返回一个 promise 对象，最终得到一个映射表，包含命令的输出、退出状态，以及任何发生的错误（分别对应于 `:out`、`:exit` 和 `:err` 键）：

```
(require '[clj-commons-exec :as exec])
```

```
(def p (exec/sh ["date"]))

(deref p)
;; -> {:exit 0, :out "Sun Dec 1 19:43:49 EST 2013\n", :err nil}
```

如果命令需要带选项或参数，只要将它们作为字符串，附加在命令向量中：

```
@(exec/sh ["ls" "-l" "/etc/passwd"])
;; -> {:exit 0
;;     :out "-rw-r--r-- 1 root   wheel  4962 May 27 07:54 /etc/passwd\n"
;;     :err nil}

@(exec/sh ["ls" "-l" "nosuchfile"])
;; -> {:exit 1
;;     :out nil
;;     :err "ls: nosuchfile: No such file or directory\n"
;;     :exception #<ExecuteException ... Process exited with an error: 1 ...)>}
```

# 讨论

到目前为止，我们并没有提到，Clojure 本身已经包含了与 `exec/sh` 相同的功能（作为 `clojure.java.shell/sh`）。读者肯定会问：既然如此，为什么要使用外部的库，而不是内建的功能？答案很简单：`clj-commons-exec` 是优秀的 Apache Commons Exec 库（http://commons.apache.org/proper/commons-exec/）的函数式封装，提供了像*管道*这样的 `clojure.java.sh` 不提供的功能。

要在多个命令之间利用管道传递数据，就使用 `clj-commons-exec/sh-pipe` 函数。就像常规的 Unix 管道那样，成对命令的 `STDOUT` 和 `STDIN` 流绑定在一起。`sh-pipe` 的 API 几乎和 `sh` 一样，唯一的不同是可以向 `sh-pipe` 传递多个命令。`sh-pipe` 的返回值是 promise 对象的列表，它们在每个子命令执行完成时得到填充：

```
(def results (exec/sh-pipe ["cat"] ["wc" "-w"] {:in "Hello, world!"}))

results
;; -> (#<core$promise$reify__6310@71eed8d: {:exit 0, :out nil, :err nil}>
;;     #<core$promise$reify__6310@7f7dc7a1: {:exit 0,
;;                                            :out "       2\n",
;;                                            :err nil}>)

@(last results)
;; -> {:exit 0, :out "       2\n", :err nil}
```

像所有合理的命令处理库一样，`clj-commons-exec` 允许配置命令执行的环境。要控制 `sh` 或 `sh-pipe` 的执行环境，就要在一个映射表中指定选项，作为这两个函数的最终参数。`:dir` 选项控制了命令执行的路径：

```
(println (:out @(exec/sh ["ls"] {:dir "/"})))
;; *out*
```

```
Applications
Library
# ...
usr
var
```

:env 和 :add-env 选项控制一些环境变量，提供给要执行的命令。:add-env 将变量附在原有的环境变量后面，而 :env 将完全取代原有的环境变量。每个选项都是一个映射表，包含变量名和值，就像 {"USER" "jeff"}：

```
@(exec/sh ["printenv" "HOME"])
;; -> {:exit 0, :out "/Users/ryan\n", :err nil}

@(exec/sh ["printenv" "HOME"] {:env {}})
;; -> {:exit 1, :out nil, :err nil, :exception #<ExecuteException ..)>}

@(exec/sh ["printenv" "HOME"] {:env {"HOME" "/Users/jeff"}})
;; -> {:exit 0, :out "/Users/jeff\n", :err nil}
```

sh 和 sh-pipe 还可以使用其他一些选项：

:watchdog

在终止命令之前，等待命令执行完成的秒数。

:shutdown

一个标志，表明子进程应该在 VM 退出时销毁。

:as-success 和 :as-successes

一个整数或整数序列，它们将被视为成功的退出码，分别对应每个命令。

:result-handler-fn

一个定制的函数，用于处理结果。

如果在 -main 函数中发起长时间的子进程，应用将挂起，直到这些进程完成。如果不想这样，就在 -main 函数的末尾直接调用 (System/exit)，强制终止你的应用。而且，对所有子进程，将 :shutdown 设为 true，确保系统干净整洁，没有恶意进程。

要检查子进程是否已经返回，又不想等到结束，就对 sh 返回的 promise 对象调用 realized? 函数（对于监控 sh-pipe 返回的 promise 对象序列的进展，这尤其有用）：

```
;; 任何运行时间长的命令
(def p (exec/sh ["sleep" "5"]))
```

```
(realized? p)
;; -> false

;; 几秒钟后……
(realized? p)
;; -> true
```

## 参阅

- 如果不需要管道或 `clj-common-execs` 的高级功能，请考虑使用 `clojure.java.shell`（http://clojure.github.io/clojure/clojure.java.shell-api.html）。

# 4.4 访问资源文件

作者：John Jacobsen，John Cromartie 和 Alex Petrov 提供了帮助

## 问题

在 Clojure 项目中，希望从 classpath 中包含一个资源文件。

## 解决方案

将资源文件放在 Leiningen 项目顶层目录的 resources/ 目录下。要继续这个实例，先用 **`lein new people`** 命令创建一个新项目。

例如，假定文件 resources/people.edn 包含下面的内容：

```
[{:first-name "John", :last-name "McCarthy", :language "Lisp"}
 {:first-name "Guido", :last-name "Van Rossum", :language "Python"}
 {:first-name "Rich", :last-name "Hickey", :language "Clojure"}]
```

将文件名称（与 resources 目录对应）传递给 `clojure.java.io/resource` 函数，得到一个 `java.io.File` 实例，然后可以自由读取（例如使用 `slurp` 函数）：

```
(require '[clojure.java.io :as io]
         '[clojure.edn :as edn])

(->> "people.edn"
     io/resource
     slurp
     edn/read-string
     (map :language))
;; -> ("Lisp" "Python" "Clojure")
```

## 讨论

资源通常用于保存各种文件，它们在逻辑上是应用的一部分，但不是代码。

资源是通过 Java 的 classpath 加载的，就像 Clojure 代码一样。在启动 Java 进程时，Leiningen 自动将 resources/ 目录放到 classpath 中。在打包时，resources/ 的内容会复制到生成的 JAR 文件的根目录中。

也可以在 project.clj 文件中使用 `:resources-paths` 键，指定另一个（或附加的）资源目录：

```
:resource-paths ["my-resources" "src/other-resources"]
```

使用基于 classpath 的资源非常方便，但也确实有一些缺点。

要注意，在 Web 应用的环境中，对资源的任何改动都可能导致全面的重新部署，因为它们全部包含在部署的 JAR 和 WAR 文件中。通常，这意味着最好只在内容完全静态时，才使用资源文件。例如，尽管可以将应用的配置文件放在 resources/ 目录下，并从这个目录加载，但这样做实际上是将它们作为应用源代码的一部分，并不适合它们的目的。对于（相对）频繁改动的资源，你可能希望将它们放在文件系统中某个已知的位置，从那里加载，而不是利用 classpath。

有时候不使用 classpath 还有其他的原因。例如，考虑网站的静态图片，如果将它们放在 Web 应用的 classpath 中，它们将由应用服务器容器（Jetty、Tomcat、JBoss 等）来提供。通常，这些应用是为提供动态 HTML 内容，而不是为较大的二进制文件而优化的。提供较大的静态文件，一般更适合让架构中的 HTTP 服务器来完成，而不是应用服务器，所以应该代理给 Apache、Nginx，或你用的其他 HTTP 服务器。或者，你甚至可能希望将它们分割开，完全通过一种独立的机制来提供它们，例如内容分发网络（CDN）。在这两种情况下，很难让 HTTP 服务器或 CDN 从应用的 JAR 文件中取出资源，所以最好是从一开始就将它们分开存储。

## 参阅

- Leiningen 的 `sample.project.clj`（https://github.com/technomancy/leiningen/blob/41f7a297b4daf4b3676048b5172a9c80c89e9266/sample.project.clj#L247），其中更详细地描述了 `:resource-paths` 选项的工作原理。
- 4.14 节“读写 Clojure 数据”。

# 4.5 复制文件

作者：Stefan Karlsson

## 问题

需要复制本地文件系统的一个文件。

## 解决方案

调用 clojure.java.io/copy，传入源文件和目标文件：

```
(clojure.java.io/copy
  (clojure.java.io/file "./file-to-copy.txt")
  (clojure.java.io/file "./my-new-copy.txt"))
;; -> nil
```

如果输入文件没找到，会抛出 java.io.FileNotFoundException：

```
(clojure.java.io/copy
  (clojure.java.io/file "./file-do-not-exist.txt")
  (clojure.java.io/file "./my-new-copy.txt"))
;; -> java.io.FileNotFoundException
```

copy 的输入参数不一定是文件，也可以是 InputStream、Reader、字节数字或字符串。这样就很容易将正在处理的数据直接复制到输出文件：

```
(clojure.java.io/copy "some text" (clojure.java.io/file "./str-test.txt"))
;; -> nil
```

如果需要，可以通过 :encoding 选项指定编码方式：

```
(clojure.java.io/copy "some text"
                      (clojure.java.io/file "./str-test.txt")
                      :encoding "UTF-8")
```

## 讨论

请注意，如果文件已经存在，它将被覆写。如果不希望这样，可以编写一个“安全”复制函数，它会捕捉所有异常，并利用可选的参数实现覆写：

```
(defn safe-copy [source-path destination-path & opts]
  (let [source (clojure.java.io/file source-path)
        destination (clojure.java.io/file destination-path)
        options (merge {:overwrite false} (apply hash-map opts))] ; ❶
    (if (and (.exists source)                                     ; ❷
             (or (:overwrite options)
                 (= false (.exists destination))))
      (try
        (= nil (clojure.java.io/copy source destination))         ; ❸
        (catch Exception e (str "exception: " (.getMessage e))))
      false)))
```

```
(safe-copy "./file-to-copy.txt" "./my-new-copy.txt")
;; -> true
(safe-copy "./file-to-copy.txt" "./my-new-copy.txt")
;; -> false
(safe-copy "./file-to-copy.txt" "./my-new-copy.txt" :overwrite true)
;; -> true
```

safe-copy 函数接受源文件和目标文件的路径，从源复制到目标。它也接受一些键 / 值对作为选项。

❶ 然后这些选项与默认值合并。在这个例子中，只有一个选项 :overwrite，但利用这个可选参数的结构，很容易添加自己的选项（例如在需要时添加 :encoding）。

❷ 在选项被处理之后，该函数检查目标文件是否已存在，如果存在，是否应该覆写。如果一切正常，它会在 try-catch 语句块中执行 copy。

❸ 在文件复制时，要注意对结果等于 nil 的检查。如果加上检查，就能确保该函数返回 Boolean 值。这会使函数用起来更方便，因为可以判断操作是否成功。

也可以用 java.io.Reader 和 java.io.Writer 作为参数，来调用 clojure.java.io/copy，还可以用流作为参数：

```
(with-open [reader (clojure.java.io/reader "file-to-copy.txt")
            writer (clojure.java.io/writer "my-new-copy.txt")]
  (clojure.java.io/copy reader writer))
```

在选择 File、Reader、Writer 或流作为输入和输出源时，需要考虑效率，这也同样适用于 copy。更多信息，请参见 4.9 节。

默认情况下，在调用 copy 时使用 1024 字节的缓冲区。这是一次从源读取并写入目标的数据量。这样一直进行到源被完整复制为止。缓冲区大小可以通过 :buffer-size 选项来更改。采用较小的数字将导致文件访问操作更多，但在内存中保留较少的数据。相反，增大缓冲区将减少文件访问的次数，但需要将更多的数据加载到内存中。

## 参阅

- clojure.java.io 的 API 文档（http://clojure.github.io/clojure/clojure.java.io-api.html）。

# 4.6 删除文件或目录

作者：Stefan Karlsson

## 问题

需要从本地文件系统中删除一个文件。

## 解决方案

使用 `clojure.java.io/delete-file` 来删除文件：

```
(clojure.java.io/delete-file "./file-to-delete.txt")
;; -> true
```

如果尝试删除一个不存在的文件，就会抛出 `java.io.IOException`：

```
(clojure.java.io/delete-file "./file-that-does-not-exist.txt")
;; -> java.io.IOException: Couldn't delete
```

如果给定的文件因为某种原因不能删除，又不希望 `delete-file` 抛出异常，那么可以将参数中的 `silently` 标志设置为 `true`：

```
(clojure.java.io/delete-file "./file-that-does-not-exist.txt" true)
;; -> true
```

## 讨论

如果你有时需要对可能的异常做一些定制的处理工作，那就应该将 `delete-file` 的调用放在 `try-catch` 语句块中：

```
(try
  (clojure.java.io/delete-file "./file-that-does-not-exist.txt")
  (catch Exception e (str "exception: " (.getMessage e))))
;; -> "exception: Couldn't delete ./file-that-does-not-exist.txt"
```

`java.io.File` 有一个 `.exists` 属性，它是一个 Boolean 值，表明文件是否存在。你可以利用这个属性和 `try-catch` 语句块，得到“安全”的删除函数。这个函数先检查参数中指定的带路径的文件是否存在，然后再尝试删除它：

```
(defn safe-delete [file-path]
  (if (.exists (clojure.java.io/file file-path))
    (try
      (clojure.java.io/delete-file file-path)
      (catch Exception e (str "exception: " (.getMessage e))))
    false))

(safe-delete "./file-that-does-not-exist.txt")
;; -> false
(safe-delete "./file-to-delete.txt")
;; -> true
```

也可以用 `clojure.java.io/delete-file` 函数来删除目录。目录必须是空的才能成功删除，所以删除目录的工具函数必须先删除该目录下的所有文件：

```
(clojure.java.io/delete-file "./dir-to-delete")
;; -> false
```

```
(defn delete-directory [directory-path]
  (let [directory-contents (file-seq (clojure.java.io/file directory-path))
        files-to-delete (filter #(.isFile %) directory-contents)]
    (doseq [file files-to-delete]
      (safe-delete (.getPath file)))
    (safe-delete directory-path)))

(delete-directory "./dir-to-delete")
;; -> true
```

`delete-directory` 将得到一个 `file-seq`，其中包含指定路径下的内容。然后它过滤出该目录下的文件，接下来删除所有文件，最后是删除目录本身。请注意对 `doall` 的调用。如果不调用 `doall`，文件的删除将是惰性的，所以在删除目录时，文件仍然存在，这样调用就会失败。

## 参阅

- `clojure.java.io` 的 API 文档（http://clojure.github.io/clojure/clojure.java.io-api.html）。
- 4.7 节“列出目录中的文件”，更详细地探讨了利用 `file-seq` 来取得目录中的文件。

# 4.7　列出目录中的文件

作者：Ryan Neufeld 和 Stefan Karlsson

## 问题

给定一个目录，希望访问其中的文件。

## 解决方案

调用内建的 `file-seq` 函数。

> 要继续这个实例，先利用以下命令，创建一些样本文件和目录（在 Linux 或 Mac 中）：
>
> ```
> $ mkdir -p next-gen
> $ touch next-gen/picard.jpg next-gen/locutus.bmp next-gen/data.txt
> ```

`file-seq` 返回一个 `java.io.File` 对象的惰性序列：

```
(def tng-dir (file-seq (clojure.java.io/file "./next-gen")))

tng-dir
```

```
;; -> (#<File  ./next-gen>
;;     #<File  ./next-gen/picard.jpg>
;;     #<File  ./next-gen/locutus.bmp>
;;     #<File  ./next-gen/data.txt>)
```

## 讨论

序列是 Clojure 的强大抽象之一。将目录层级作为一个序列，就可以利用 `map` 和 `filter` 这样的函数来操纵文件和目录。

例如，假设只想选择目录层级中的文件（不包含目录）。你可以定义一个函数，取得文件和目录的序列，并用 `java.io.File` 对象的 `.isFile` 属性对它们进行过滤：

```
(defn only-files
  "Filter a sequence of files/directories by the .isFile property of
  java.io.File"
  [file-s]
  (filter #(.isFile %) file-s))

(only-files tng-dir)
;; -> (#<File ./next-gen/data.txt>
;;     #<File ./next-gen/locutus.bmp>
;;     #<File ./next-gen/picard.jpg>)
```

如果想显示所有文件的名称呢？定义一个 `names` 函数，对文件序列映射 `.getName` 属性，然后组合使用 `only-files` 和 `names`，得到目录下的文件名列表：

```
(defn names
  "Return the .getName property of a sequence of files"
  [file-s]
  (map #(.getName %) file-s))

(-> tng-dir
    only-files
    names)
;; -> ("data.txt" "locutus.bmp" "picard.jpg")
```

## 参阅

- `File` 类的文档（http://docs.oracle.com/javase/7/docs/api/java/io/File.html），了解 `File` 对象的属性和方法的完整列表。
- 将这些技术与一些工具库结合，如 Google Guava 的 `Files` 类（http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/io/Files.html）或 Apache Commons 的 `FilenameUtils` 类（http://commons.apache.org/proper/commons-io/javadocs/api-1.4/org/apache/commons/io/FilenameUtils.html），从文件的序列抽象中得到更大的好处。

# 4.8 文件的内存映射

作者：Alan Busby

## 问题

希望用内存映射来访问大文件，就像它完全载入内存一样，但实际上没有加载整个文件。

## 解决方案

利用 `clj-mmap` 库（https://github.com/thebusby/clj-mmap），它包装了 Java 的 NIO（新 I/O）库的内存映射功能。

开始之前，请在项目依赖关系中加入 `[clj-mmap "1.1.2"]`，或用 `lein-try` 开始 REPL：

```
$ lein try clj-mmap
```

要读取 UTF-8 编码的文本文件的开头和末尾 N 个字节，请用 `get-bytes` 函数：

```
(require '[clj-mmap :as mmap])

(with-open [file (mmap/get-mmap "/path/to/file/file.txt")]
  (let [n-bytes      10
        file-size     (.size file)
        first-n-bytes (mmap/get-bytes file 0 n-bytes)
        last-n-bytes (mmap/get-bytes file (- file-size n-bytes) n-bytes)]
    [(String. first-n-bytes "UTF-8")
     (String. last-n-bytes "UTF-8")]))
```

要覆写文本文件开头 N 个字节，请用 `put-bytes`：

```
(with-open [file (mmap/get-mmap "/path/to/file/file.txt")]
  (let [bytes-to-write (.getBytes "New text goes here" "UTF-8")
        file-size      (.size file)]
    (if (> file-size
           (alength bytes-to-write))
      (mmap/put-bytes file bytes-to-write 0))))
```

## 讨论

内存映射或 POSIX 标准的 mmap，是利用操作系统的虚拟内存来进行文件 I/O 的方法。通过将文件映射到应用的内存空间，缓冲区之间的复制行为减少了，I/O 性能增加了。

内存映射的文件对于大文件、结构化的二进制文件尤其有用，对于文本文件避免 Java 的 `String` 开销也很有用。

虽然 Clojure 可以方便地直接调用 Java NIO 的功能，但 NIO 在处理超过 2GB 的文件时特别困难。`clj-mmap` 封装了这种复杂性，但它没有提供 NIO 的全部功能。在需要的时候，仍可以通过互操作性来调用 NIO Java API。

## 参阅

- `mmap` 的维基百科页面（http://en.wikipedia.org/wiki/Mmap）。
- `clj-mmap` 的 GitHub 代码库（https://github.com/thebusby/clj-mmap）。

# 4.9　读写文本文件

作者：Stefan Karlsson

## 问题

希望读写本地文件系统中的文本文件。

## 解决方案

用内建的 `spit` 函数将字符串写入文件：

```
(spit "stuff.txt" "my stuff")
```

用内建的 `slurp` 函数读取文件的内容：

```
(slurp "stuff.txt")
;; -> "all my stuff"
```

如果需要，可以利用 `:encoding` 选项来指定编码方式：

```
(slurp "stuff.txt" :encoding "UTF-8")
;; -> "all my stuff"
```

用 `spit` 函数的 `:append true` 选项，将数据添加到已有文件的后面：

```
(spit "stuff.txt" "even more stuff" :append true)
```

要一行一行地读取文件，而不是一次将所有内容装入内存，请使用 `java.io.Reader` 和 `line-seq` 函数：

```
(with-open [r (clojure.java.io/reader "stuff.txt")]
  (doseq [line (line-seq r)]
    (println line)))
```

要将大量的数据写入文件，又不想将数据变成一个字符串，请使用 `java.io.Writer`：

```
(with-open [w (clojure.java.io/writer "stuff.txt")]
  (doseq [line some-large-seq-of-strings]
    (.write w line)
    (.newLine w)))
```

## 讨论

如果使用 :append，文本将添加在文件末尾。要打印的字符串加上 "\n"，在每行末加上换行。文本文件中所有行末都要有换行，包括最后一行：

```
(defn spitn
  "Append to file with newline"
  [path text]
  (spit path (str text "\n") :append true)
```

spit 和 slurp 与字符串一起使用，实现一次性处理文件的全部内容，并在读写完成后关闭文件。如果需要读写很多数据，使用流式 API 更有效（在内存和时间方面），比如 java.io.Reader 或 java.io.Writer，因为它们不需要在内存中重新实现文件的内容。

但是，如果使用 writer 对象和流，有一点很重要：即清空所有 writer 对象，写入底层的流，以确保数据真正写入，资源得到释放。with-open 宏在执行完它的内容后，将清空并关闭它绑定的流。

> 尤其要注意，如果基于流的惰性序列在序列实现之前，底层的流被关闭了，它将抛出错误。甚至在使用 with-open 时，也有可能返回未实现的惰性序列，with-open 宏无法知道流仍需要打开，所以不管怎样都会关闭流，导致序列无法实现。

一般来说，最好不要让基于流的惰性序列超出流打开的代码范围。如果一定要这样做，必须非常小心，只要惰性序列还需要读取，就要确保实现惰性序列所需的资源一直打开。通常，这样做需要手工记录哪些流仍然打开，而不是依赖于 try/finally 或 with-open 代码块。

## 参阅

- 4.14 节“读写 Clojure 数据”。
- java.io.Reader（http://docs.oracle.com/javase/7/docs/api/java/io/Reader.html）和 java.io.Writer（http://docs.oracle.com/javase/7/docs/api/java/io/Writer.html）的文档。

# 4.10　使用临时文件

作者：Alan Busby

## 问题

希望在本地文件系统中使用临时文件。

## 解决方案

使用 Java 内建的 `java.io.File` 类的 `createTempFile` 静态方法，在 JVM 的默认临时目录中创建临时文件，参数是文件名和扩展名：

```
(def my-temp-file (java.io.File/createTempFile "filename" ".txt"))
```

然后可以写入该临时文件，就像用任何其他 `java.io.File` 实例一样：

```
(with-open [file (clojure.java.io/writer my-temp-file)]
  (binding [*out* file]
    (println "Example output.")))
```

## 讨论

在与其他程序交互时，如果它们使用基于文件的 API，那么临时文件常常很有用。使用 `createTempFile` 很重要，它能确保临时文件放在文件系统的合适位置。使用的操作系统不同，这个位置也不同。

要取得创建的临时文件的完整路径和文件名，就调用：

```
(.getAbsolutePath my-temp-file)
```

可以用 `File.deleteOnExit` 方法，指明在 JVM 退出时，自动删除该临时文件：

```
(.deleteOnExit my-temp-file)
```

请注意，在 JVM 退出前，该文件不会被删除，而且如果进程崩溃或非正常退出，它可能也不会被删除。好的做法是随时删除那些不再需要的临时文件：

```
(.delete my-temp-file)
```

## 参阅

- `java.io.File` 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/io/File.html）。

# 4.11　在任意位置读写文件

作者：John Jacobsen

## 问题

希望在文件的任意位置读写，而不是顺序读写。

## 解决方案

要打开一个（可能非常大的）文件进行随机访问，请用 Java 的 RandomAccessFile。seek 到你期望的位置，然后利用各种 write 方法，在这个位置写入数据。

例如，要生成一个 1 GB 的文件，除末尾的整数 1234 之外，全部以 0 填充：

```
(import '[java.io RandomAccessFile])

(doto (RandomAccessFile. "/tmp/longfile" "rw")
  (.seek (* 1000 1000 1000))
  (.writeInt 1234)
  (.close))
```

对一个“标准的”Java 文件对象调用 length 方法，表明文件的大小是正确的：

```
(require '[clojure.java.io :refer [file]])
(.length (file "/tmp/longfile"))

;; -> 1000000004
```

（也可以直接对 RandomAccessFile 调用 length。）

在 Clojure 中，从适当的位置读取值和写入是非常类似的。同样，对 RandomAccessFile 调用 seek，然后使用合适的 read 方法：

```
(let [raf (RandomAccessFile. "/tmp/longfile" "r")
      _ (.seek raf (* 1000 1000 1000))
      result (.readInt raf)]
  (.close raf)
  result)

;; -> 1234
```

## 讨论

用这种方式写文件，默认会以 0 填充。它可能会被 JVM 实现和底层的操作系统当成是“稀疏文件”，从而获得更高的读写效率。

利用 Unix 的 od 程序检查我们刚创建的文件，通过命令行进行 16 进制导出，可以看到该文件由 0 组成，1234 在末尾。

```
$ od -Ad -tx4 /tmp/longfile
0000000          00000000     00000000     00000000     00000000
```

```
*
1000000000          d2040000
1000000004
```

在字节偏移 1000000000 处，可以看到值 `d2040000`，这是 1234 的 16 进制，采用的是大端（big-endian）整数表示法。（Java 整数默认是大端表示法。这意味着最高位的字节存放在最低位的地址中。）

## 参阅

- 4.14 节“读写 Clojure 数据”，了解读取整个文件的更多信息。
- `java.io.RandomAccessFile` 的 API 文 档（http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html）。
- Unix 的 `od` 命令（http://en.wikipedia.org/wiki/Od_(Unix)）。

# 4.12 并行文件处理

作者：Edmund Jackson

## 问题

希望逐行转换一个文本文件，但想用到所有的 CPU 内核，而不把文件载入到内存。

## 解决方案

解决这个问题的捷径就是对 `line-seq` 返回的序列应用 `pmap`：

```
(require ['clojure.java.io :as 'jio])

(defn pmap-file
  "Process input-file in parallel, applying processing-fn to each row
  outputting into output-file"
  [processing-fn input-file output-file]
  (with-open [rdr (jio/reader input-file)
              wtr (jio/writer output-file)]
    (let [lines (line-seq rdr)]
      (dorun
       (map #(.write wtr %)
            (pmap processing-fn lines))))))

;; 调用这个的例子
(def accumulator (atom 0))

(defn- example-row-fn
  "Trivial example"
  [row-string]
```

```
  (str row-string "," (swap! accumulator inc) "\n"))

;; 调用它
(pmap-file example-row-fn "input.txt" "output.txt")
```

## 讨论

除了 map 或 dorun 这样的基本 Clojure 结构之外，这个例子使用的关键函数是 line-seq 和 pmap。

line-seq 利用 java.io.BufferedReader 的实例（该实例由 clojure.java.io/reader 返回），返回一个字符串的惰性序列。每个字符串是输入文件中的一行。分行时以什么作为新行的标识，这由 JVM 的选项 line.separator 来决定，它将根据不同的平台来设定。具体来说，在 Windows 上是回车加换行，在 Linux 或 Mac OS X 这样的类 Unix 系统上，只是一个换行。

pmap 与 map 功能相同，将一个函数作用于序列中的每个元素，得到由返回值构成的惰性序列。不同之处在于，它在执行映射函数时，对集合中的每个元素启用一个独立的线程（最多到一定的数目，这与系统的 CPU 数有关）。如果值还没有准备好，实现该序列的线程将阻塞。

通过将工作分派到多个 CPU 内核上并行执行，pmap 能实现可观的性能改进，但它并非神通广大。具体来说，它为了实现多线程的调度，导致了一定的协作开销。通常，如果执行的是重量级操作，映射函数是计算密集型的，协作开销就值得，这种方法就能提供最大的好处。而对于能够快速完成的简单函数（例如对原生数据的基本操作），协作开销就有点得不偿失了，在这种情况下，pmap 实际上可能比 map 慢很多。

设想是使用 pmap 并行地处理文件行序列。但是，接着就需要让每个被处理的行依次通过 (map #(.write wtr %) ...)，以确保这些行每次写入一行（或者将 write 放到处理函数中，看看会发生什么情况）。最后，因为是惰性序列，所以需要先实现它们的副作用，然后再退出 with-open 语句块，否则在你希望求值时，文件已经关闭了。这是通过调用 dorun 实现的。

这里有一些警告。首先，尽管输出文件中行的次序与输入文件相同，但处理的次序是不保证相同的。其次，这个过程很快就变成 I/O 密集型，因为所有的写入都在一个线程中完成，所以除非处理函数非常耗 CPU，否则速度提升可能不及预期。最后，pmap 在分配工作时并非完美，所以实际速度提升程度也许不能像你期望的那样对应系统中处理器的数目。

pmap 方式还有另一点不足，实际文件是串行读取的，利用了一个 java.io.Reader。如果处理任务比读取要耗时得多，仍然可以获得可观的提速，但对于轻量级任务，瓶颈可能是读取文件本身，在这种情况下，并行处理对总体运行时间就没有太多帮助（甚至可能更糟）。

## 参阅

- 4.13 节“带归约的并行文件处理”，探讨了一个类似的方法，利用内存映射来并行读取文件（也用了 Clojure 的归约程序取得更高效率）。

# 4.13　带归约的并行文件处理

作者：Edmund Jackson

## 问题

希望对文件使用 Clojure 的归约程序，实现并行处理，而不把文件载入到内存。

## 解决方案

使用 Iota 库（https://github.com/thebusby/iota），以及 Clojure 归约库的 `filter`、`map` 和 `fold` 函数，它们在 `clojure.core.reducers` 命名空间中。要继续这个实例，请在项目依赖关系中加入 `[iota "1.1.1"]`，或用 `lein-try` 开始 REPL：

```
$ lein try iota
```

例如，要统计一个很大的文件中的单词个数：

```
(require '[iota                  :as io]
         '[clojure.core.reducers :as r]
         '[clojure.string        :as str])


;; 统计单词个数函数
(defn count-map
  "Returns a map of words to occurence count in the given string"
  [s]
  (reduce (fn [m w] (update-in m [w] (fnil (partial inc) 0)))
          {}
          (str/split s #" ")))

(defn add-maps
  "Returns a map where each key is the sum of vals of that key in m1 and m2."
  ([] {}) ;; 必要的基础，用于折叠时的结合
  ([m1 m2]
     (reduce (fn [m [k v]] (update-in m [k] (fnil (partial + v) 0))) m1 m2)))


;; 主文件处理
(defn keyword-count
  "Returns a map of the word counts"
  [filename]
  (->> (iota/seq filename)
```

```
    (r/filter identity)
    (r/map count-map)
    (r/fold add-maps)))
```

# 讨论

Iota 从本地文件系统的文件中创建序列。不像 `file-seq` 得到的纯顺序式惰性序列，Iota 得到的序列是针对 Clojure 的归约库优化的，该库利用了 Java 的 Fork/Join 工作窃取（work-stealing）框架[1]，以提供高效的并行处理。

`keyword-count` 函数首先创建了一个可归约的序列，包含文件中的行，并过滤掉空行（利用 `identity` 函数从序列中消除 `nil` 值）。然后它并行地应用 `count-map` 函数，最后汇聚结果，用 `add-maps` 函数来折叠。

`r/filter` 和 `r/map` 函数做的事情与不针对归约库的相应函数一样，区别在于性能以及归约库分解和组合操作的方式。它们也返回可归约的序列，能被归约库中的其他操作有效地利用。

`r/fold` 是归约库的核心函数，其基本形式在功能上与内建的 `reduce` 函数非常类似。给定一个函数和一个可归约的集合，它返回一个值，即对集合中的每个元素应用折叠函数并累加的值。

但是，与普通的 `reduce` 不同，这里不保证执行的次序，这也是为什么 `fold` 不接受一个初始值作为参数的原因。那样做没有意义，因为计算可以同时从几个地方"开始"，并发进行。这意味着传递给 `fold` 的函数（在传递一个函数时）必须能够接受 `0` 个参数，因为对提供的函数进行无参数的调用，将被作为每个计算分支的种子值。

如果需要更多灵活性，`fold` 还允许同时指定 `reduce` 函数和 `combine` 函数作为独立的参数。具体如何工作与归约函数的工作方式密切相关，所以完整的解释超出了本实例的讨论范围。更多信息，请参见 `fold` 函数的 API 文档（http://clojure.github.io/clojure/clojure.core-api.html#clojure.core.reducers/fold），以及 Clojure 网站的 Reducer 页面（http://clojure.org/reducers）。

### 关于归约库

归约库是一个并行计算框架，目的是极为高效地并行处理。完整解释归约库的工作原理超出了这个实例的讨论范围（来龙去脉请参见 Clojure 网站上关于引入归约库的博客帖子 http://clojure.com/blog/2012/05/08/reducers-a-library-and-model-for-collection-processing.html）。但简单来说，归约库通过以下两种方式来提升性能。

注 1：更多信息，参见 Java 指南中关于 Fork/Join 和 work stealing 的部分（http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html）。

- 它们能组合操作。在逻辑许可时，归约框架将能组合的操作合并成一个操作。例如，前面执行 `filter` 的代码和接着执行 `map` 的代码。Clojure 的标准 `filter` 和 `map` 将实现一个中间序列：`filter` 将产生一个序列，然后交给 `map`。但在归约库中，会将它们组合成一个 `map+filter` 操作（如果可能的话），一次执行。
- 它们利用了被归约数据的内部树状数据结构。普通序列生来就是顺序化的（这不奇怪），因为它们的高性能操作就是每次从头上取出元素，所以很难有效地在各个元素间分配工作。但是，归约库注意到了 Clojure 持久数据结构的内部结构，并利用它来有效地分配数据处理工作。

在底层，Iota 使用了 Java NIO 库，为处理的文件提供内存映射视图，实现高效的随机访问。Iota 也考虑到归约框架，Iota 的序列构造方式让归约库能有效地分配处理工作。

## 参阅

- Iota 的 GitHub 代码库（https://github.com/thebusby/iota）。
- NIO 文档（http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html）。

# 4.14 读写Clojure数据

作者：John Cromartie

## 问题

需要存取磁盘上的 Clojure 数据结构。

## 解决方案

用 `pr-str` 和 `spit` 来序列化少量数据：

```
(spit "data.clj" (pr-str [:a :b :c]))
```

用 `read-string` 和 `slurp` 读取少量数据：

```
(read-string (slurp "data.clj"))
;; -> [:a :b :c]
```

用 `pr` 向流中高效地写入大型数据结构：

```
(with-open [w (clojure.java.io/writer "data.clj")]
  (binding [*out* w]
    (pr large-data-structure)))
```

用 `read` 从流中高效地读取大型数据结构：

```
(with-open [r (java.io.PushbackReader. (clojure.java.io/reader "data.clj"))]
  (binding [*read-eval* false]
    (read r)))
```

## 讨论

在 Clojure 中，代码就是数据。在运行时，你可以使用编程语言从文件加载代码所使用的读取程序，这使得这个任务变得比较简单。虽然这通常是将数据持久到磁盘上的好方法，但也应该注意一些问题。

> **读取、安全和 edn**
>
> `read` 函数只适合从信任的来源读取数据。这是因为 Clojure 的读取程序不是为安全而设计的，它不能保证安全，也不能避免副作用。将 `*read-eval*` 绑定到 `false` 只是小小的安全保护措施。如果需要从不信任的来源读取 Clojure 数据结构（即不是你自己写下的数据），就要看 `clojure.edn` 库。
>
> edn（可扩展数据表示法）是 Clojure 的数据结构序列化格式的一种规范，有多种实现，所以可以用作传输和持久格式，并被所有程序使用（无论程序是用何种语言写成的），就像 XML 或 JSON 一样。`clojure.edn` 库是 edn 的 Clojure 实现。
>
> 它的工作方式很像 Clojure 的读取程序和写入程序，但它提供了额外的安全保证，这是 Clojure 的读取程序做不到的。对于外部或不信任的输入，总是应该使用它。

如果数据量非常大，简单情况下的 `slurp` 和 `spit` 就不可用了，因为会在内存中一下子创建非常大的字符串。例如，序列化 100 万随机数（由 `rand` 创建），将得到 18 MB 的文件，在读写时将消耗更多的内存：

```
(spit "data.clj" (pr-str (repeatedly 1e6 rand)))
;; -> OutOfMemoryError Java heap space ...
```

但是，如果你知道只要处理少量的数据，这种方法就非常合适。对于加载配置数据或其他类型的简单结构，它是一种好方法。

用流读写更为高效，因为它是带缓冲的输入和输出，同时处理一些字节的数据[2]。

除了读写文件中的单个数据结构外，还可以将更多的数据结构添加到同一个文件中，以后作为序列读取：

```
(spit "data.clj" (prn-str [1 2 3]))
(spit "data.clj" (prn-str [:a :b :c]) :append true)
;; data.clj now contains two serialized structures
```

---

注 2：关于管理流的注意事项，参见 4.9 节。

随着时间的推移在文件后面加上少量数据很有用处，例如作为事件或事务日志。

但是，要从一个字符串中读取多个对象，`read-string` 是不够的。要从流中读取一系列对象，必须连续调用 `read`，直到结束。

```
(defn- read-one
  [r]
  (try
    (read r)
    (catch java.lang.RuntimeException e
      (if (= "EOF while reading" (.getMessage e))
        ::EOF
        (throw e)))))

(defn read-seq-from-file
  "Reads a sequence of top-level objects in file at path."
  [path]
  (with-open [r (java.io.PushbackReader. (clojure.java.io/reader path))]
    (binding [*read-eval* false]
      (doall (take-while #(not= ::EOF %) (repeatedly #(read-one r)))))))
```

## 参阅

- 4.4 节“访问资源文件”。
- 4.15 节“在配置文件中使用 edn”。
- 4.17 节“读取 Clojure 数据时处理未知的带标签字面值”。

# 4.15 在配置文件中使用edn

作者：Luke VanderHart

## 问题

希望使用像 Clojure 那样的数据字面量来配置应用。

## 解决方案

用保存在 edn 文件中的 Clojure 数据结构来定义一个映射表，包含需要的配置项。

例如，一个应用需要知道它自己的主机名和关系数据库的连接信息，它的 edn 配置可能是这样：

```
{:hostname "localhost"
 :database {:host "my.db.server"
            :port 5432
            :name "my-app"
```

```
              :user "root"
              :password "s00p3rs3cr3t"}}
```

利用 edn 读取程序，将这段数据读入 Clojure 映射表的基本函数是很简单的：

```
(require '[clojure.edn :as edn])

(defn load-config
  "Given a filename, load & return a config file"
  [filename]
  (edn/read-string (slurp filename)))
```

调用新定义的 `load-config` 函数会得到一个配置映射表，可以像其他映射表一样，在应用中传递和使用。

## 讨论

从前面的代码中可以看到，得到包含配置数据的映射表的过程很简单。更有趣的问题是，得到配置映射表后如何使用。一般来说有两种思路。

第一种思路注重开发的方便，让配置映射表作为背景环境，在整个应用中随时可以访问。通常会设置一个全局的 var，包含该配置。

但是，由于某些原因，这样做容易出问题。首先，很难在不同环境下覆写默认的配置文件，例如测试环境，或是在同一个 JVM 中运行两个配置不同的系统。（可以利用线程局部绑定来绕过，但这将很快导致代码混乱。）

更重要的是，使用全局配置意味着读取配置的函数（在相当大的应用中，这是大多数函数）*不是纯的*。在 Clojure 中，这意味着放弃了很多。纯 Clojure 代码的主要好处是它的局部透明性，函数的行为只要看它的参数和代码就能确定。但是，如果每个函数都读取全局变量，这就变得困难得多。

另一种思路是在所有需要的地方，显式传递配置，就像所有其他参数一样。因为配置文件通常是在应用启动时提供的，所以配置常常在 `-main` 函数中建立，再传递到所有需要的地方。

这听起来很痛苦，对每个函数多传一个参数确实也让人有点气恼。但这样做让代码在很大程度上能自我解释，应用的哪些部分依赖于配置变得十分清楚。这也使得运行时修改配置更简单，在测试场景中提供不同的配置也更容易。

### 使用多个配置文件

在配置应用时，一种常见的模式就是有几类不同的配置项。某些配置字段比较稳定，在相同的环境中，不会在不同的应用实例之间变化。它们常常和应用源代码一起，提交到源代

码版本控制中。

另一些配置项也相当稳定，但出于安全考虑，不能提供到源代码版本控制中。这样的例子包括数据库口令或安全 API 的令牌，它们最好是放在独立的配置文件中。还有一些配置字段（如 IP 地址）通常在每个部署实例中都完全不同，最好是独立指定它们，与稳定的配置字段分开。

处理这种不同有一个技巧，就是使用多个配置文件，每个配置文件处理一种不同类型的考虑，然后将它们合并为一个配置映射表，传递给应用。通常会用一个简单的 `deep-merge` 函数：

```
(defn deep-merge
  "Deep merge two maps"
  [& values]
  (if (every? map? values)
    (apply merge-with deep-merge values)
    (last values)))
```

这会合并两个映射表，如果 `values` 都是映射表，就合并它们。如果 `values` 不全是映射表，那么第二个“胜出”，出现在结果映射表中。

然后，重写配置加载程序，接受多个配置文件，合并在一起：

```
(defn load-config
  [& filenames]
  (reduce deep-merge (map (comp edn/read-string slurp)
                          filenames)))
```

对两个独立的 edn 配置文件（config-public.edn 和 config-private.edn）运用这种方法，得到一个合并的映射表。

- config-public.edn:

```
{:hostname "localhost"
 :database {:host "my.db.server"
            :port 5432
            :name "my-app"
            :user "root"}}
```

- config-private.edn:

```
{:database {:password "s3cr3t"}}

(load-config "config-public.edn" "config-private.edn")
;; -> {:hostname "localhost", :database {:password "s3cr3t",
;;      :host "my.db.server", :port 5432, :name "my-app", :user "root"}}
```

要注意，在两个配置文件中都有的值，将取 `load-config` 的最右参数文件中的值。

### 不同环境下的不同配置

如果系统运行在多种环境中，你可能希望根据当前运行的环境来变动配置。例如，在开发环境中希望连接到本地数据库，在产品环境中希望连接到产品数据库。

可以利用 Leiningen 的 profiles 功能来实现这一点。通过为项目配置中的每个文件提供不同的 `:resource-paths` 选项，可以变更每种环境下读入的配置文件 [3]：

```
(defproject my-great-app "0.1.0-SNAPSHOT"
  {;; ...
  :profiles {:dev {:resource-paths ["resources/dev"]}
             :prod {:resource-paths ["resources/prod"]}}})
```

项目配置与前面的类似，现在你可以创建两个不同的配置，具有相同的文件名，即 resources/dev/config.edn 和 resources/prod/config.edn:

- `resource/dev/config.edn`:

  ```
  {:database-host "localhost"}
  ```

- `resources/prod/config.edn`:

  ```
  {:database-host "production.example.com"}
  ```

如果你打算继续自己操作，请将 `load-config` 函数添加到项目的一个命名空间中：

```
(ns my-great-app.core
  (:require [clojure.edn :as edn]))

(defn load-config
    "Given a filename, load & return a config file"
    [filename]
    (edn/read-string (slurp filename)))
```

现在，应用加载的配置将取决于项目运行时采用哪个文件：

```
# "dev" 是 Leiningen 的默认文件之一
$ lein repl
user=> (require '[my-great-app.core :refer [load-config]])
user=> (load-config (clojure.java.io/resource "config.edn"))
{:database-host "localhost"}
user=> (exit)

$ lein trampoline with-profile prod repl
user=> (require '[my-great-app.core :refer [load-config]])
user=> (load-config (clojure.java.io/resource "config.edn"))
{:database-host "production.example.com"}
```

注 3：要继续操作，请用 **`lein new my-great-app`** 创建你自己的项目。

## 参阅

- 4.4 节“访问资源文件”。
- 4.16 节“将记录作为 edn 值发布”。
- 4.17 节“读取 Clojure 数据时处理未知的带标签字面值”。
- Leiningen profiles 指南（https://github.com/technomancy/leiningen/blob/master/doc/TUTORIAL.md#profiles）。

# 4.16 将记录作为edn值发布

作者：Steve Miner

## 问题

希望将 Clojure 记录当成 edn 值使用，但 edn 不支持记录。

## 解决方案

可以利用 `tagged` 库，将记录作为 edn 带标签的字面值来读入或输出。开始之前，请在项目依赖关系中加入 `[com.velisco/tagged "0.3.0"]`，或用 `lein-try` 开始 REPL：

```
$ lein try com.velisco/tagged
```

要扩展 Clojure 内建的 `print-method` 多重方法，以带标签的形式来打印记录，就要利用 `miner.tagged/pr-tagged-record-on` 辅助函数，扩展针对该记录的 `print-method`：

```
(require '[miner.tagged :as tag])

(defrecord SimpleRecord [a])

(def forty-two (->SimpleRecord 42))

(pr-str forty-two)
;; -> "#user.SimpleRecord{:a 42}" ;; 不幸的是，不是正确的 edn 值

(defmethod print-method user.SimpleRecord [this w]
  (tag/pr-tagged-record-on this w))

(pr-str forty-two)
;; -> "#user/SimpleRecord {:a 42}"
```

现在，就可以利用 edn 带标签的字面值格式，在 `pr-str` 和 `miner.tagged/read-string` 之间来回转换记录：

```
(tag/read-string (pr-str forty-two))
;; -> #user/SimpleRecord {:a 42}

(= forty-two
   (tag/read-string (pr-str forty-two)))
;; -> true
```

但是，edn 读取程序仍不知道如何解析这些带标签的值。为了支持这一行为，在用 `edn` 读取值时，用 `miner.tagged/tagged-default-reader` 作为 `:default` 选项：

```
(require '[clojure.edn :as edn])

(edn/read-string {:default tag/tagged-default-reader}
                 (pr-str {:my-record forty-two}))
;; -> {:my-record #user/SimpleRecord {:a 42}}
```

## 讨论

edn 格式非常好，因为它覆盖了 Clojure 数据类型中一个有用的子集，让高保真的数据传输变得很容易。不幸的是，它不支持记录。但这很容易修正，根据名字，edn 是一种可扩展的格式。我们只需要提供标签风格的打印输出（`#tag <value>`）和适当的读取程序。`tagged` 库使得这些任务非常容易。

从前面的例子可以看出，Clojure 默认的记录打印值比较像，但并不是 edn 期望的带标签的格式。

虽然 Clojure 对 `SimpleRecord` 打印出 `"#user.SimpleRecord{:a 42}"`，但是 edn 需要的是带标签的字符串，就像 `"#user/SimpleRecord {:a 42}"`。函数 `miner.tagged/pr-tagged-record-on` 知道如何以这种格式输出记录（到一个 `java.io.Writer`）。用这个函数来扩展 Clojure 的 `print-method` 多重方法，就能确保 Clojure 总是以带标签的格式输出记录。

要读入这些值，就需要告诉 edn 读取程序如何解析新的记录标签。根据设计，`tagged` 库提供了 `miner.tagged/tagged-default-reader` 函数，可以用来扩展 edn，读取你的记录标签。如果 edn 读取程序不能解析标签，它就尝试用 `:default` 选项指定的函数来处理标签。通过指定 `tagged-default-reader` 作为它的 `:default` 选项，edn 读取程序就能够正确地解析带标签的记录值。

## 参阅

- 4.17 节“读取 Clojure 数据时处理未知的带标签字面值”，了解 `:default` 选项的更多信息。
- GitHub 上的 edn 扩展数据注释（https://github.com/edn-format/edn）。

# 4.17 读取Clojure数据时处理未知的带标签字面值

作者：Steve Miner

## 问题

希望读取 Clojure 数据（采用 edn 格式），其中可能包含未知的带标签字面值。

## 解决方案

使用 `clojure.edn/read` 或 `clojure.edn/read-string` 的 `:default` 选项：

```
(require 'clojure.edn)

(defrecord TaggedValue [tag value])

(defn read-preserving-unknown-tags [s]
  (clojure.edn/read-string {:default ->TaggedValue} s))

(read-preserving-unknown-tags "#my.example/unknown 42")
;; -> #user.TaggedValue{:tag my.example/unknown, :value 42}
```

## 讨论

edn 格式对 Clojure 数据类型的重要子集定义了打印表示形式，通过带标签的字面值来提供扩展性。读取 edn 数据最好的方法就是 `clojure.edn/read` 或 `clojure.edn/read-string`。这些函数分别从流或字符串中消费 edn 格式的数据，返回处理过的 Clojure 数据。

两个函数都接受一个 `opts` 映射表，它允许你控制读取时的几个选项。对于已经知道的标签，可以提供一个 `:readers` 映射表，定义定制的读取程序。可以用这个映射表覆写内建类型的行为，它们是在 `clojure.core/default-data-readers` 中定义的：

```
;; 创建定制的读取程序
(clojure.edn/read-string {:readers {'inc-this inc}}
                         "#inc-this 1")
;; -> 2

;; 覆写内建的读取程序
;; 之前……
(clojure.edn/read-string "#inst \"2013-06-08T01:00:00Z\"")
;; -> #inst "2013-06-08T01:00:00.000-00:00"

;; 之后……
(clojure.edn/read-string {:readers {'inst str}}
                         "#inst \"2013-06-08T01:00:00Z\"")
;; -> "2013-06-08T01:00:00Z"
```

这个解决方案中探讨了 `:default` 选项，它非常适合处理未知的标签。不论遇到什么未知的标签或值，你提供的函数都会被调用，两个参数分别是标签和值。

如果没有向 read 提供 :default 选项，读到未知的标签就会抛出 Runti meException：

```
(clojure.edn/read-string "#blow-up boom")
;; -> RuntimeException No reader function for tag blow-up ...
```

对于大多数应用，读取未知的标签是一个错误，所以抛出异常是适当的。但是，有时候也许只是进入另一个处理阶段，可能需要保留“未知的”标签。

利用 defrecord 定义的工厂函数，很容易捕捉到未知的读取程序字面量。TaggedValue 工厂的参数次序与 :default 数据读取程序的规格说明是匹配的，这很方便。

TaggedValue 记录保留了基本信息，以备将来使用。因为所有读入的信息都被保留下来，所以甚至可以用原来带标签的字面量的格式再次打印该值。

```
(defmethod print-method TaggedValue [this ^java.io.Writer w]
   (.write w "#")
   (print-method (:tag this) w)
   (.write w " ")
   (print-method (:value this) w))

;; 现在，TaggedValue 将输出为原来带标签的字面量
(read-preserving-unknown-tags "#my.example/unknown 42")
;; -> #my.example/unknown 42
```

**clojure.core/read**

edn 读取程序并不是一直就有的。在以前，如果想读取 Clojure 数据，可以使用两个内建的读取函数 clojure.core/read 或 clojure.core/read-string。这两个函数的目的是从“信任的来源”读取代码或数据。

因为这些函数能执行代码[1]，所以永远不应该使用 clojure.core 的读取程序从不信任的来源读取数据。这意味着用户的数据、远程服务器（即使属于你），或者几乎是任何有关系的地方都不行（当然，这有点极端，但我们希望你安全）。

如果环境“确实”是安全的，或者绝对需要执行一些代码，那就务必使用 clojure.core 的读取程序。但是，在设置选项的接口方面，这些读取程序确实与 clojure.edn 的读取程序不同。不是传入 opts，而是要改变各种动态绑定，才能调节读写程序的行为。例如，*default-data-reader-fn* 决定了核心函数如何处理未知的标签。更多信息请参见 *data-readers* 和 *read-eval*（http://clojure.github.io/tools.reader/）。这就是说，对于读入数据，使用 edn 函数通常更好[2]。

1 这实际上是一个特征，因为它们是语言用来执行代码的函数。

2 Clojure 邮件列表中的主题“ANN: NEVER use clojure.core/read or read-string for reading untrusted data”（https://groups.google.com/forum/#!topic/clojure/YBkUaIaRaow）讨论了关于 clojure.core 读取程序脆弱性的更多信息。

## 参阅

- edn：GitHub 上的扩展数据注释（https://github.com/edn-format/edn）。
- 4.14 节“读写 Clojure 数据”，以及 4.16 节“将记录作为 edn 值发布”。

# 4.18　从文件中读取属性

作者：Tobias Bayer

## 问题

需要读取属性文件，并访问其中的键值对。

## 解决方案

最直接的方法就是利用 Java 的互操作性，使用内建的 `java.util.Properties` 类（http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html）。`java.util.Properties` 实现了 `java.util.Map` 接口，Clojure 可以像其他映射表一样轻松地使用。

下面的例子是要读入的属性文件 fruitcolors.properties：

```
banana=yellow
grannysmith=green
```

用文件中的内容填充一个 `Properties` 实例是很容易的，只要用它的 `load` 方法，并传入一个 `java.io.Reader` 实例就行。该实例是通过 `clojure.java.io` 命名空间得到的：

```
(require '[clojure.java.io :refer (reader)])

(def props (java.util.Properties.))

(.load props (reader "fruitcolors.properties"))
;; -> nil

props
;; -> {"banana" "yellow", "grannysmith" "green"}
```

除了通过互操作性来使用内建的 `Properties` API 之外，也可以用 `propertea` 库，用更简单、更符合 Clojure 习惯的方式来访问属性文件。

在项目的 project.clj 文件中添加依赖关系 `[propertea "1.2.3"]`，或用 `lein-try` 启动 REPL：

```
$ lein try propertea 1.2.3
```

然后读取属性文件，访问它的键值对：

```
(require '[propertea.core :refer (read-properties)])

(def props (read-properties "fruitcolors.properties"))

props
;; -> {:grannysmith "green", :banana "yellow"}
(props :banana)
;; -> "yellow"
```

## 讨论

虽然直接使用 java.util.Properties 更容易想到，也不需要额外的依赖关系，但是 propertea 确实提供了一些便利。它返回一个确实不可改变的 Clojure 映射表，而不是 java.util.Map。虽然两者在 Clojure 中都很好用，但如果打算做进一步的操作或更新，不可改变的映射表可能更合适。

更重要的是，propertea 将所有键字符串转换成了关键字，在 Clojure 的映射表中，比用字符串作为键更为常见。

而且，propertea 还有其他一些特征，比如能够将值解析为数字或布尔类型，并提供默认值。

在默认情况下，propertea 的 read-properties 函数将所有属性值都作为字符串。请考虑以下属性文件，包含值为整数和布尔类型的键：

```
intkey=42
booleankey=true
```

通过提供包含 :parse-int 和 :parse-boolean 选项的列表，可以强制将这些属性解析为对应的类型：

```
(def props (read-properties "other.properties"
                            :parse-int [:intkey]
                            :parse-boolean [:booleankey]))

(props :intkey)
;; -> 42

(class (props :intkey))
;; -> java.lang.Integer

(props :booleankey)
;; -> true

(class (props :booleankey))
;; -> java.lang.Boolean
```

有时候属性文件可能不包含某个键值对，但你希望设置合理的默认值：

```
(def props (read-properties "other.properties" :default [:otherkey "awesome"]))

(props :otherkey)
;; -> "awesome"
```

也可以要求必须具有某些属性。如果期望的属性不在属性文件中，会抛出异常：

```
(def props (read-properties "other.properties" :required [:otherkey]))
;; -> java.lang.RuntimeException: (:otherkey) are required ...
```

## 参阅

- `propertea` GitHub 代码库（https://github.com/jaycfields/propertea）。
- `Properties` API 文档（http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html）。

# 4.19　读写二进制文件

作者：John Jacobsen

## 问题

需要读写一些二进制数据。

## 解决方案

用 Java 的 `BufferedInputStream`、`BufferedOutputStream` 和 `ByteBuffer` 类，直接处理二进制数据。

## 讨论

虽然在纯 Clojure 中读写文本文件很容易（例如用 `slurp` 和 `spit`），但写入二进制数据就需要通过 Java 互操作。

Clojure 的 `output-stream` 包装了 Java 的 `BufferedOutputStream` 对象。`BufferedOutputStream` 有一个 `write` 方法，它接受 Java 字节数组。下面的代码向 /tmp/zeros 文件写入 1000 个 0（字节）：

```
(require '[clojure.java.io :refer [file output-stream input-stream]])

(with-open [out (output-stream (file "/tmp/zeros"))]
  (.write out (byte-array 1000)))
```

要读取这些字节，使用对应的 `input-stream` 函数，它包装了 `BufferedInputStream`：

```
(with-open [in (input-stream (file "/tmp/zeros"))]
  (let [buf (byte-array 1000)
        n (.read in buf)]
    (println "Read" n "bytes.")))

;;=> Read 1000 bytes.
```

写入 0 和读取固定长度的块显然不太有趣。我们希望用一些实际的内容来准备字节数组。准备字节数组的常见方法是使用 ByteBuffer，用来自不同类型的数据填充它。假定我们要用下面的格式写入“字符串”。

(1) 版本号（byte，例子中是 66）。
(2) 字符串的长度（大端 int）。
(3) 字符串对应的字节（这里是“hello world”）。

下面的函数利用 ByteBuffer 作为中介，将字节“包装”好放入数组中：

```
(import '[java.nio ByteBuffer])

(defn prepare-string [strdata]
  (let [strlen (count strdata)
        version 66
        buflen (+ 1 4 (count strdata))
        bb (ByteBuffer/allocate buflen)
        buf (byte-array buflen)]
    (doto bb
      (.put (.byteValue version))
      (.putInt (.intValue strlen))
      (.put (.getBytes strdata))
      (.flip)          ;; 准备好 bb 用于读取
      (.get buf))
    buf))

(prepare-string "hello world")
;;=> #<byte[] [B@5ccab0e8>
(into [] (prepare-string "hello world"))
;;=> [66 0 0 0 11 104 101 108 108 111 32 119 111 114 108 100]
```

然后用这种格式写数据就简单了：

```
(with-open [out (output-stream "/tmp/mystring")]
  (.write out (prepare-string "hello world")))
```

要取回数据，ByteBuffer 提供了一种方法，从字节流（数组）中解包出多个类型：

```
(defn unpack-buf [n buf]
  (let [bb (ByteBuffer/allocate n)]
    (.put bb buf 0 n)                    ;; 用数组内容填充 ByteBuffer
    (.flip bb)                           ;; 准备读取
```

```
    (let [version (.get bb 0)]
      (.position bb 1)                    ;; 跳过版本字节
      (let [buflen (.getInt bb)
            strbytes (byte-array buflen)] ;; 准备缓冲区来保存字符串数据...
        (.get bb strbytes)                ;; ……读取数据。
        [version buflen (apply str (map char strbytes))]))))


(with-open [in (input-stream "/tmp/mystring")]
  (let [buf (byte-array 1024)
        n (.read in buf)]
    (unpack-buf n buf)))

;=> [66 11 "hello world"]
```

请注意，对于写入和读取，对 ByteBuffer 的 flip 操作都将位置重置到缓冲区开始的地方，准备读取和写入。

## 参阅

- ByteBuffer 在 Java 的 NIO 库中扮演了重要角色，关于它的更多细节，参见 Java NIO 文档（http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html）或 Ron Hitchens 的著作 Java NIO（http://shop.oreilly.com/product/9780596002886.do, O'Reilly）。
- Clojure 库 bytebuffer（https://github.com/geoffsalmon/bytebuffer）提供了轻便的、更符合习惯的 ByteBuffer 操作包装。
- 更新一点的 Buffy 库（https://github.com/clojurewerkz/buffy）提供了相关的 Netty ByteBuffer 的包装。
- 最后，Gloss 库（https://github.com/ztellman/gloss）提供了一个 DSL，来读取和写入二进制数据流（不论基于文件还是基于网络）。

# 4.20　读写CSV数据

作者：Jason Whitlark

## 问题

需要读写 CSV 数据。

## 解决方案

使用 clojure.data.csv/read-csv，从 String 或 java.io.Reader 中惰性读取 CSV 数据：

```
(clojure.data.csv/read-csv "this,is\na,test" )
;; -> (["this" "is"] ["a" "test"])
```

```
(with-open [in-file (clojure.java.io/reader "in-file.csv")]
  (doall
    (clojure.data.csv/read-csv in-file)))
;; -> (["this" "is"] ["a" "test"])
```

使用 clojure.data.csv/write-csv 向 java.io.Writer 写入 CSV 数据：

```
(with-open [out-file (clojure.java.io/writer "out.csv")]
            (clojure.data.csv/write-csv out-file [["this" "is"] ["a" "test"]]))
;; -> nil
```

## 讨论

Clojure.data.csv 库让处理 CSV 变得很容易。要记住 read-csv 是惰性的，如果要迫使它马上读取数据，就需要用 doall 包装 read-csv 调用。

在读取时，可以改变分隔符和引号定义符，默认的分别是 \ 和 \"。但指定分隔符必须用字符，而不是字符串：

```
(csv/read-csv "this$-is $-\na$test" :separator \$ :quote \-)
;; -> (["this" "is $"] ["a" "test"])
```

在写入时，就像 read-csv，你可以配置分隔符、引号和换行（选择 :lf（默认）还是 :cr+lf），以及 quote? 谓词函数，它接受一个集合，返回 true 或 false，表明字符串表示是否需要加引号：

```
(with-open [out-file (clojure.java.io/writer "out.csv")]
            (clojure.data.csv/write-csv out-file [["this" "is"] ["a" "test"]]
                                        :separator \$ :quote \-))
;; -> nil
```

要将 CSV 输出作为字符串记录下来，就用 with-out-str，并写入 *out*：

```
(with-out-str (csv/write-csv *out* [["this" "is"] ["a" "test"]]))
;; -> "this,is\na,test\n"
```

## 参阅

- clojure.data.csv 的 GitHub 代码库（https://github.com/clojure/data.csv）。

# 4.21　读写压缩文件

作者：John Cromartie

## 问题

希望读取或写入 gzip 压缩的文件（即 .gz 文件）。

## 解决方案

用 `java.util.zip.GZIPInputStream`（http://docs.oracle.com/javase/7/docs/api/java/util/zip/GZIPInputStream.html）包装普通输入流，得到解压的数据：

```
(with-open [in (java.util.zip.GZIPInputStream.
                (clojure.java.io/input-stream
                 "file.txt.gz"))]
  (slurp in))
```

用 `java.util.zip.GZIPOutputStream`（http://docs.oracle.com/javase/7/docs/opi/java/util/zip/GZIPOutputStream.html）包装普通输出流，在写入时压缩数据：

```
(with-open [w (-> "output.gz"
                  clojure.java.io/output-stream
                  java.util.zip.GZIPOutputStream.
                  clojure.java.io/writer)]
  (binding [*out* w]
    (println "This will be compressed on disk.")))
```

## 讨论

基于 DEFLATE 算法的 gzip，是类 Unix 系统中的常见压缩方式，大量用于 Web 的压缩。它特别适合压缩文本，能大幅压缩源代码或 Clojure 和 JSON 的数据。

许多 Clojure 的 I/O 函数接受所有类型的 Java 流。`GZIPInputStream` 只是包装了另一个输入流，并尝试对原来的流进行解压缩。输出的行为也是类似的。

通过包装一个普通输入流，比如 `clojure.java.io/input-stream` 的返回值，你可以将它传递给 `slurp` 或 `line-seq`（或其他接受输入流作为参数的函数），方便地读取所有解压缩的内容。

也可以利用这种技巧逐行读取大型的压缩文件，或者读取 `pr` 和 `pr-str` 输出的 Clojure 格式。还可以用类似的方式解压来自任何其他流的数据，例如，基于网络套接字或字节数组的流。

将输出流绑定到 `*out*`，我们就可以用 `println`、`pr` 等函数，一次向流中输入少量的数据，这些数据在流关闭时，将压缩到磁盘上。

可以用几乎同样的方法来写入 ZIP 压缩格式的数据，只要使用 `java.util.zip.ZipInputStream`（http://docs.oracle.com/javase/7/docs/api/java/util/zip/ZipInputStream.html）

和 `java.util.zip.ZipOutputStream`（http://docs.oracle.com/javase/7/docs/api/java/util/zip/ZipOutputStream.html）类。

## 参阅

- 4.14 节“读写 Clojure 数据”，探讨了从磁盘文件读取 Clojure 数据。
- `GZIPInputStream` 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/util/zip/GZIPInputStream.html）。

# 4.22 处理XML数据

作者：Stefan Karlsson

## 问题

需要读写 XML 数据。

## 解决方案

将文件传递给 `clojure.xml/parse`，得到一个 Clojure 映射表，表示了 XML 文件的结构。

例如，要读取下面的文件：

```
<simple>
  <item id="1">First</item>
  <item id="2">Second</item>
</simple>
```

利用 `clojure.xml/parse`：

```
(require '[clojure.xml :as xml])
(clojure.xml/parse (clojure.java.io/file "simple.xml"))
;; -> {:tag :simple, :attrs nil, :content [
;;     {:tag :item, :attrs {:id "1"}, :content ["First"]}
;;     {:tag :item, :attrs {:id "2"}, :content ["Second"]}]}
```

如果希望以节点序列的方式来读取 XML 文件，就将 XML 的映射表传递给 `xml-seq` 函数，该函数属于 `clojure.core` 命名空间：

```
(xml/xml-seq (clojure.xml/parse (clojure.java.io/file "simple.xml")))
```

`xml-seq` 返回节点的树状序列，即由每个节点构成的序列，从根开始，然后对文档进行深度优先遍历。

要写入 XML 文件，就将 XML 结构映射表传递给 `clojure.xml/emit`。`emit` 将该 XML `spit`

到当前绑定的输出流（*out*），所以要写入文件，要么将 *out* 绑定到该文件的输出流，要么用 with-out-str 宏将输出流记录到字符串中，然后用 spit 写入文件：

```
(spit "test.xml" (with-out-str (clojure.xml/emit simple-xml-map)))
```

## 讨论

处理 XML 数据时，可以像处理其他映射表一样。下面例子中的函数，对给定的 id 和文件，将解析该文件，找到 id 属性与参数一致的节点：

```
(defn get-with-id [id xml-file]
  (for [node (xml-seq (clojure.xml/parse xml-file))
        :when (= (get-in node [:attrs :id]) id)]
    (:content node)))

(get-with-id "2" simple-xml)

;; -> (["Second"])
```

要修改 XML，只要对这种 Clojure 数据表示形式使用普通的映射表操作函数。

如果要处理大量的 XML 结构，你可以考虑使用 zipper。zipper 是一种纯函数式数据结构，用于方便高效地实现树状结构（如 XML）的导航和修改。

zipper 是一个深奥的话题，全面的讨论超出了本实例的范围。但请看一下 clojure.data.zip 库（http://clojure.github.io/data.zip/）的文档，其中提供了解释和例子，说明如何用它们高效地处理 XML。

## 参阅

- 4.9 节“读写文本文件”。
- clojure.zip 命名空间的 API 文档（http://clojure.github.io/clojure/clojure.zip-api.html）。

# 4.23　读写JSON数据

作者：Stefan Karlsson

## 问题

需要读写 JSON 数据。

## 解决方案

用 clojure.data.json/read-str 函数读取 JSON 字符串，成为 Clojure 数据：

```
(require '[clojure.data.json :as json])
(json/read-str "[{\"name\":\"Stefan\",\"age\":32}]")
;; -> [{"name" "Stefan", "age" 32}]
```

要将数据写回成 JSON，请使用 clojure.data.json/write-str 函数，以原来的 Clojure 数据作为参数：

```
(json/write-str [{"name" "Stefan", "age" 32}])
;; -> "[{\"name\":\"Stefan\",\"age\":32}]"
```

## 讨论

除了读写字符串之外，clojure.data.json 也提供了 read 和 write 函数，分别与 java.io.Reader 和 java.io.Writer 对象合作。除了 reader/writer 参数不同之外，这两个函数与使用字符串的对应函数具有相同的参数和选项：

```
(with-open [writer (clojure.java.io/writer "foo.json")]
  (json/write [{:foo "bar"}] writer))

(with-open [reader (clojure.java.io/reader "foo.json")]
  (json/read reader))
;; -> [{"foo" "bar"}]
```

由于 JavaScript 的类型比较简单，所以 JSON 表示法的保真度比 Clojure 数据低很多。因此你可能会发现，需要调整键或值的解释方式。一个常见的例子就是将 JSON 的字符串键转换成合适的 Clojure 关键字。通过 :key-fn 选项，可以对每个处理的键应用一个函数：

```
;; 在读取时修改键

(json/read-str "{\"name\": \"Stefan\"}")
;; -> {"name" "Stefan"}

(json/read-str "{\"name\": \"Stefan\"}" :key-fn keyword)
;; -> {:name "Stefan"}

;; 在写入时修改键

(json/write-str {:name "Stefan"})
;; -> "{\"name\":\"Stefan\"}"

(json/write-str {:name "Stefan"} :key-fn str)
;; -> "{\":name\":\"Stefan\"}" ; 注意多出来的\:
```

你也许想控制值的解释方式。请用 :value-fn 选项来指定值读写的方式。你提供的函数在调用时会传入两个参数，键和对应的值：

```
;; 正确地读取 UUID 值
(defn str->uuid [key value]
  (if (= key :uuid)
```

```
    (java.util.UUID/fromString value)
    value))

(clojure.data.json/read-str
  "{\"name\": \"Stefan\", \"uuid\": \"51674ca0-eadc-4a5b-b9fb-67b05d5a71b7\"}"
  :key-fn keyword
  :value-fn str->uuid)
;; -> {:name "Stefan", :uuid #uuid "51674ca0-eadc-4a5b-b9fb-67b05d5a71b7"}

;; 类似地，写入 UUID 值
(defn uuid->str [key value]
  (if (= key :uuid)
    (str value)
    value))

(clojure.data.json/write-str
  {:name "Stefan", :uuid #uuid "51674ca0-eadc-4a5b-b9fb-67b05d5a71b7"}
        :value-fn uuid->str)
;; -> "{\"name\":\"Stefan\",\"uuid\":\"51674ca0-eadc-4a5b-b9fb-67b05d5a71b7\"}"
```

你可能已经猜到，如果同时提供 :key-fn 和 :value-fn，值函数总是会在键函数之后调用。

不必说，:key-fn 和 :value-fn 选项也适用于 write 和 read 函数。

## 参阅

- 4.14 节“读写 Clojure 数据”，探讨了读写 edn（Clojure）数据。
- clojure.data.json 的 API 文档（http://clojure.github.io/data.json/）探讨了关于读写的更多内容。本实例未讨论的选项包括 read 的 :eof-error?、:eof-value 和 :bigdec，以及 write 的 :escape-unicode 和 :escape-slash。

# 4.24 生成PDF文件

作者：Dmitri Sotnikov

## 问题

希望利用一些数据生成 PDF。

例如，有一系列映射表，可能是通过 clojure.java.jdbc 查询得到的，你希望生成一份 PDF 报告。

## 解决方案

使用 clj-pdf 库来生成报告。

开始之前，请在项目依赖关系中加入 `[clj-pdf "1.11.6"]`，或用 `lein-try` 开始 REPL：

```
$ lein try clj-pdf
```

为了说明，假设我们希望展现一个向量，其中包含以下雇员记录：

```
(def employees
 [{:country "Germany",
   :place "Nuremberg",
   :occupation "Engineer",
   :name "Neil Chetty"}
  {:country "Germany",
   :place "Ulm",
   :occupation "Engineer",
   :name "Vera Ellison"}])
```

利用 `clj-pdf.core/template` 宏，创建一个模板来展现每条记录：

```
(require '[clj-pdf.core :as pdf])

(def employee-template
 (pdf/template
   [:paragraph
    [:heading (.toUpperCase $name)]
    [:chunk {:style :bold} "occupation: "] $occupation "\n"
    [:chunk {:style :bold} "place: "] $place "\n"
    [:chunk {:style :bold} "country: "] $country
    [:spacer]]))

(employee-template employees)
;; -> ([:paragraph [:heading "NEIL CHETTY"]
;;        [:chunk {:style :bold} "occupation: "] "Engineer" "\n"
;;        [:chunk {:style :bold} "place: "] "Nuremberg" "\n"
;;        [:chunk {:style :bold} "country: "] "Germany" [:spacer]]
;;      [:paragraph [:heading "VERA ELLISON"]
;;        [:chunk {:style :bold} "occupation: "] "Engineer" "\n"
;;        [:chunk {:style :bold} "place: "] "Ulm" "\n"
;;        [:chunk {:style :bold} "country: "] "Germany"
;;        [:spacer]])
```

利用上面的模板和数据，调用 `clj-pdf.core/pdf` 来生成 PDF：

```
(pdf/pdf [{:title "Employee Table"}
          (employee-template employees)]
          "employees.pdf")
```

你会在运行项目 /REPL 的目录下，找到 employees.pdf 文件，如图 4-1 所示。

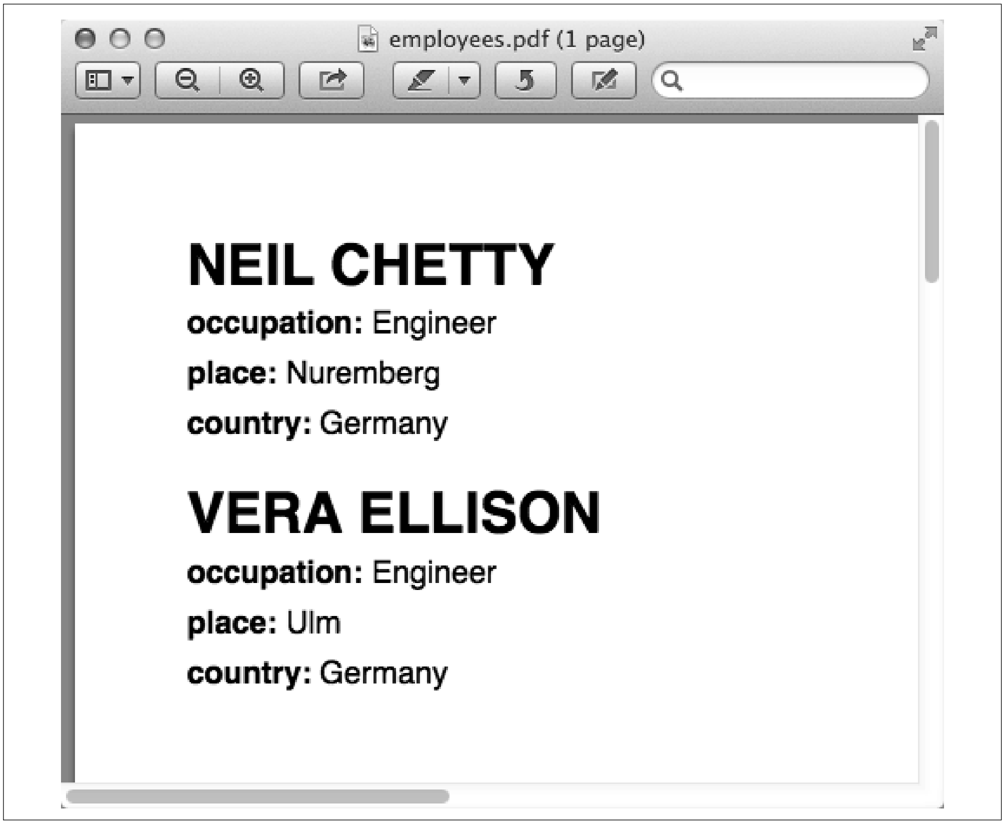



图 4-1：employees.pdf

## 讨论

clj-pdf 基于 iText 和 JFreeChart 库。模板语法受到了流行的 Hiccup HTML 模板引擎的启发。

在模板中，$ 用来表示将由动态内容替换的位置。如果用一个映射表来填充模板，每个替换占位符（$name）都会用映射表中对应键的值来填充（:name 键的值）。

除了替换简单的值，也可以对这些值进一步处理。employee-template 的 :heading 部分就是这么做的，它调用了 (.toUpperCase $name)。在 clj-pdf 中，文档表示为一个向量，其中包含元数据的映射表，然后是内容。内容又可以包含字符串、向量，或向量的集合。

一个非常简单的 PDF：

```
(pdf/pdf [{:title "Hello World"} "Hello, World."] "hello-world.pdf")
```

在背后，一组内容自动展开：

```
;; 这是 * 一组 * 段落……
(pdf [{} [[:paragraph "foo"] [:paragraph "bar"]]] "document.pdf")

;; 等价于这些 * 独立 * 段落
(pdf [{} [:paragraph "foo"] [:paragraph "bar"]] "document.pdf")
```

除了普通字符串，每个内容元素都表示为一个向量。向量中的第一个元素是关键字的类型，后面跟的都是内容。clj-pdf 的一些类型包括 :paragraph、:phrase、:list 和 :table：

```
[:heading "Lorem Ipsum"]
[:line]
[:list "first item"
      "second item"
      "third item"]
[:paragraph "I'm a paragraph"]
[:phrase "some text here"]
[:table
   ["foo" "bar" "baz"]
   ["foo1" "bar1" "baz1"]
   ["foo2" "bar2" "baz2"]]
```

某些元素接受可选的风格元数据。可以用一个映射表跟在类型参数后面（作为向量的第二个元素），提供这种风格信息：

```
[:paragraph {:style :bold} "this text is bold"]

[:chunk {:style :bold
         :size 18
         :family :helvetica
         :color [0 234 123]}
 "some large green text"]
```

一个元素的内容也可以包含其他元素（就像 HTML 文档），应用于父元素的风格将被子元素继承：

```
[:paragraph "some content"]

[:paragraph {:style :bold}
 "Some bold text"
 [:phrase [:chunk "even more"] "bold text"]]
```

就像层叠样式表（CSS），子元素可以扩展或覆写父元素的风格，指定自己的风格：

```
[:paragraph
 {:style :bold}
 "Bold words"
 [:phrase {:color [0 255 221]} "Bold AND teal!"]]
```

图像可以用 :image 元素嵌入在文档中。图像的内容可以是 java.net.URL、java.awt.Image、字节数组、Base64 字符串，或表示 URL 或文件的字符串：

```
[:image "my-image.jpg"]
[:image "http://clojure.org/space/showimage/clojure-icon.gif"]
```

比页面大的图像将自动缩小，以适合页面。

## 参阅

- 要获取使用 `clj-pdf` 的更多信息，包括元素类型和图表功能的完整清单，参见 `clj-pdf` 的 GitHub 代码库（https://github.com/yogthos/clj-pdf）。

# 4.25　生成带可滚动文本的GUI窗口

作者：John Jacobsen，最初由 John Walker 提交

## 问题

希望创建并显示一个 GUI 窗口。

## 解决方案

虽然 Java 的 Swing 库是创建 Java GUI 最常见的方式（至少是在桌面应用中），但 Seesaw 库才是用 Clojure 创建 GUI 的最佳工具，它包装了 Swing，提供了更符合习惯的函数式接口。

要继续这个实例，先用 `lein-try` 启动 REPL：

```
$ lein try seesaw
```

Swing 实现了“可编程的观感”：各种窗口小部件的外观和行为都可以修改，但常见的做法是设置与所处的平台匹配，以便获得最好的易用性。在 Seesaw 中，设置本地观感是通过 `native!` 函数来实现的：

```
(require '[seesaw.core :refer [native! frame show! config!
                               pack! text scrollable]])

(native!)
;; -> nil
```

要创建窗口对象，就用 `frame`（它在背后生成了 Swing 对象 `JFrame`）：

```
(frame :title "Lyrical Clojure" :content "Hello World")
;; -> #<JFrame$Tag$a79ba523 seesaw.core.proxy$javax.swing.JFrame$Tag$a79ba523
;;    [frame0,0,22,0x0,invalid,hidden,layout=java.awt.BorderLayout,
;;    title=Lyrical Clojure,resizable,normal,
;;    defaultCloseOperation=HIDE_ON_CLOSE,
```

```
;;    rootPane=javax.swing.JRootPane[,0,0,0x0,invalid,
;;    layout=javax.swing.JRootPane$RootLayout,
;;    alignmentX=0.0,alignmentY=0.0,border=,flags=16777673,maximumSize=,
;;    minimumSize=,preferredSize=],rootPaneCheckingEnabled=true]>
```

虽然创建了窗体，但什么也没有显示。要显示窗体（如图 4-2），就用 show!：

```
(def f (frame :title "Lyrical Clojure"))

(show! f)
;; -> #<JFrame$Tag$a79ba523 [...]>
```




图 4-2：简单窗口

## 讨论

创建了窗口后，可以设置它的大小，添加内容和滚动条，详情见下文。

### 添加内容

可以用 config! 来改变窗口的属性：

```
(config! f :content "Actual content!")
;; -> #<JFrame$Tag$a79ba523 [...]>
```

结果如图 4-3 所示。




图 4-3：包含基本内容的窗口

### 改变窗口大小

可以在创建时指定窗口的大小：

```
(def f (frame :title "Lyrical Clojure" :width 300 :height 150))
;; -> #<JFrame$Tag$a79ba523 [...]>
```

但是，替代做法通常是对得到的窗体对象调用 pack!，根据其内容来指定宽度和高度属性：

```
(-> f pack! show!)
;; -> #<JFrame$Tag$a79ba523 [...]>
```

### 添加可滚动内容

现在向窗口添加一些文本，摘录莎士比亚的十四行诗：

```
(def sonnet-text (->> "http://www.gutenberg.org/cache/epub/1041/pg1041.txt"
                      slurp
                      (drop 20000)
                      (take 4000)
                      (apply str)))
```

内容太长，当前窗口无法容纳（参见图 4-4）。

```
(config! f :content sonnet-text)
;; -> #<JFrame$Tag$a79ba523 [...]>
```

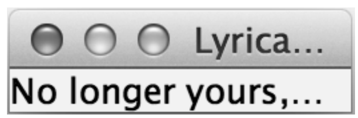



图 4-4：包含的文字超出空间的窗口

一般会再次调用 pack!，调整窗口大小，适应新的内容。但在大多数屏幕上，都不能放下全部的内容，所以明确地设置窗口大小，并添加滚动条，如图 4-5 所示。

```
(.setSize f 400 400)
(config! f :content (scrollable (text :multi-line? true
                                      :text sonnet-text
                                      :editable? false)))
```

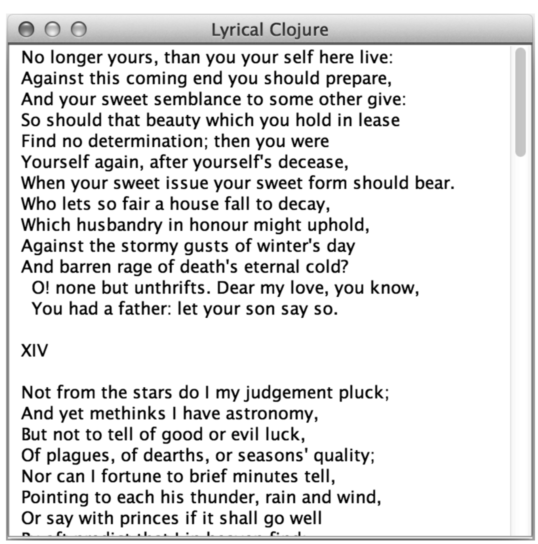



图 4-5：带滚动条的较大窗口

`text` 函数的 `:multi-line?` 选项将选择 `JTextArea` 作为底层对象，而不是 `JTextField`（`JTextArea` 用于多行文本，`JTextField` 用于单行文本）。`:editable?` 指明你不希望允许用户编辑这段文本（也许是因为他们不太可能改进莎士比亚的原作）。像其他创建窗口小部件的 Seesaw 函数一样，`text` 还有一些选项，最好通过 API 文档（http://daveray.github.io/seesaw/）来学习。

在 Clojure 中，Seesaw 的函数返回 Java 对象，总是可以直接用 Java 方法来操作它们。例如，我们用到了 `frame` 返回的 `JFrame` 对象的 `.setSize` 方法。这种互操作性提供了强大的能力，但代价是程序员的负担变重了，他们不仅要使用 Seesaw API，而且常常要用到底层 Swing API 的某些功能。

Seesaw 支持各种 GUI 任务，包括创建菜单、显示文本和图像、滚动条、单选框、复选框、多区域窗口、拖放等等。除了已经有的几十本 Swing 的书，你可以很容易写一整本书来讨论 Seesaw。这个实例只是一个起点，将来可以进一步研究 Seesaw 库。

## 参阅

- Seesaw GitHub 代码库（https://github.com/daveray/seesaw）。
- Marc Loy 等人所著 *Java Swing*, 2nd ed。(O’Reilly, http://shop.oreilly.com/product/9780596004088.do）。

第 5 章

# 网络I/O和Web服务

## 5.0 简介

我们建造的每个系统都必须与其他系统通信的趋势越来越明显[1]。如果不通过某种网络与其他计算机通信，我们几乎做不了任何事情。

本章探讨了你能想到的所有标准的远程通信模式（HTTP、TCP、UDP 等），也探讨了一些新兴技术[2]，如面向消息的架构等。

## 5.1 发出HTTP请求

作者：John Cromartie

### 问题

希望发出简单的 HTTP GET 或 POST 请求。

### 解决方案

用 `slurp` 发出简单的 HTTP GET 请求：

注 1：实际上，“没办法回避：你正在建造分布式系统”（http://queue.acm.org/detail.cfm?id=2482856）。

注 2：正如澳大利亚歌曲作者 Peter Allen 精辟的说法：“Everything old is new again”（所有老的东西又新了）。

```
(slurp "http://example.com")
;; -> "<!doctype html>\n<html>\n<head>\n    <title>Example Domain</title> ...
```

用 clj-http 库来发出 GET、POST 和其他请求，带上特定的参数或请求头，处理重定向或其他特殊情况，或取得响应的具体细节。

要继续本实例，请在项目依赖关系中加入 [clj-http "0.7.7"]，或用 lein-try 开始 REPL：

```
$ lein try clj-http
```

用 clj-http.client/get 发出 GET 请求：

```
(require '[clj-http.client :as http])

(:status (http/get "http://clojure.org"))
;; -> 200

(-> (http/get "http://clojure.org")
    :headers
    (get "server"))
;; -> "nginx"

(-> (http/get "http://www.amazon.com/")
    :cookies
    keys)
;; -> ("session-id" "session-id-time" "x-wl-uid" "skin")
```

GET 和 POST 请求都可以带参数。用 clj-http.client/post 发出 POST 请求：

```
(http/get "http://google.com/" {:query-params {:q "clojure"}})
;; -> {:status 200 ...}

(http/post "http://example.com" {:form-params {:username "joecoder"
                                               :password "il0v3clojure"}})
;; -> {:status 200 ...}
```

甚至可以用 :multipart 选项来上传文件，就像通过 Web 浏览器的 HTML 表格一样。

## 讨论

slurp 能发出 HTTP GET 请求，这是因为它的参数被传递给 clojure.java.io/reader，后者能正确处理并打开 URL 字符串。如果要向行为良好的 URL 发起快速的 HTTP GET，这就足够了。不幸的是，slurp 的用处仅此而已。除了其他限制之外，它对于 HTTP 重定向也不能正确处理。

clj-http 是非常灵活的 Clojure 库，它包装了非常强大的 Apache HttpComponents 库（https://hc.apache.org/）。它的功能包括针对其他 HTTP 动词的方便的函数，如 PUT 和 DELETE，收发 cookie、请求 / 响应头和其他请求元数据，利用流、文件或字节数组来读

写数据，等等。请参考 GitHub 代码库（https://github.com/dakrone/clj-http），了解更多不同的选项，以及更多的例子。

如果你构造的产品系统依赖于外部的服务，可能要考虑用 Netflix 的 Hystrix 库（https://github.com/Netflix/Hystrix）来包装 HTTP 调用，让应用的容错性更好，更有弹性。Hystrix 提供了 Clojure 绑定（https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-clj），可以用来包装网络调用，更容易地管理涉及外部服务的复杂失效场景。

## 参阅

- `clj-http` 的 GitHub 代码库（https://github.com/dakrone/clj-http）。
- 关于异步 HTTP 调用的信息，请参见 5.2 节“执行异步 HTTP 请求”。
- 如果构建涉及外部服务的产品系统，请考虑 Hystrix（https://github.com/Netflix/Hystrix）及其 Clojure 绑定（https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-clj），以便处理复杂的失效场景。

# 5.2 执行异步HTTP请求

作者：Alan Busby 和 Ryan Neufeld

## 问题

希望执行异步 HTTP 请求。

## 解决方案

使用 HTTP Kit（http://www.http-kit.org/），它是高性能的、事件驱动的 HTTP 客户端 / 服务器库。

在开始之前，请在项目依赖关系中加入 `[http-kit "2.1.12"]`，或用 `lein-try` 开始 REPL：

```
$ lein try http-kit
```

用 `org.httpkit.client` 的任何一个 HTTP 动词函数，来执行异步 HTTP 请求。在这些函数的基本形式中，它们会返回一个 promise 对象，你可以用 `deref` 来等待，或用 `@` 便捷读取：

```
(require '[org.httpkit.client :as http])

(def response (http/get "http://example.com"))

;; 一段时间后……

(:status @response)
```

```
;; -> 200

;; 或者，用 deref 来指定超时毫秒数和一个值
(deref response 2000 nil)
;; -> {:opts {:url "http://example.com", :method :get}
;;     :body "..."
;;     :headers {:content-type "text/html", :content-length "1270" ...}
;;     :status 200}
```

## 讨论

执行 HTTP 请求时，多半时间花在建立连接和等待服务器响应上。异步请求让应用在等待数据传输时，能继续工作。

在这方面，HTTP Kit 提供了高度并发的 Web 服务器和强大的 HTTP 客户端。对于异步请求，它提供了回调和 promise，也提供了长连接和备用的 SSL 引擎，处理未签名的 SSL 证书。

`org.httpkit.client` 命名空间定义了许多 HTTP 方法的异步版本，包括 `get`、`delete`、`head`、`post`、`put`、`options` 和 `patch`。每个动词都源自于 `org.httpkit.client/request`，它定了一个公共的接口。指定方法的异步请求发出，返回一个 promise 对象。当请求完成时，这个 promise 将被结果或响应填充。

所有 `request` 函数都接受一个可选的选项映射表，其中可以指定一些键，如 `:query-params`、`:post-params` 或 `:headers`。这些函数也允许指定回调函数，在请求完成时调用：

```
(http/get "http://example.com"
          {:timeout 1000 ;; ms
                 :query-params {:search "value"}}
          (fn [{:keys [status headers body error]}]
            (if error
              (binding [*out* *err*]
                (println "Failed with, " error))
              (println body))))
;; -> #<core$promise$reify__6310@582e6c93: :pending>
;; *out*
;; <html>
;; <head>
;;   <title>Example Domain</title>
;; ...
```

## 参阅

- 关于发出普通的、非异步的 HTTP 请求，请参见 5.1 节。
- HTTP Kit 在很大程度上受到了 `clj-http` API（https://github.com/dakrone/clj-http）的启发，关于这个库的更多信息，参见 5.1 节。

# 5.3 发出Ping请求

作者：Jason Webb

## 问题

希望 ping 一个 IP 地址，检查它是否可访问。

## 解决方案

使用 `java.net.InetAddress` 类，检查该地址是否 `isReachable`：

```
(.isReachable (java.net.InetAddress/getByName "oreilly.com") 5000)
;; -> true
```

## 讨论

如果能得到正确的授权，使用 `isReachable` 的效果就很好。在典型的类 Unix 实现中，需要用 `sudo` 来启动 Clojure 实例，才能发出实际的 ICMP ping 包。否则，标准连接会尝试端口 7，这常常会被防火墙封掉。Javadoc（http://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html#isReachable(int)）中有更多相关信息。

在 ping 另一台机器时，常常需要计时。可以用一个函数 `timed-ping` 来包装 `.isReachable` 调用，返回每次 ping 的时间：

```
(defn timed-ping
  "Time an .isReachable ping to a given domain"
  [domain timeout]
  (let [addr (java.net.InetAddress/getByName domain)
        start (. System (nanoTime))
        result (.isReachable addr timeout)
        total (/ (double (- (. System (nanoTime)) start)) 1000000.0)]
    {:time total
     :result result}))

(timed-ping "oreilly.com" 5000)
;; -> {:time 88.07, :result true}
```

## 参阅

- InetAddress/isReachable 的文档（http://docs.oracle.com/javase/7/docs/api/java/net/InetAddress.html#isReachable(int)）。

# 5.4 取得并解析RSS数据

作者：Osbert Feng

## 问题

需要解析 RSS 数据。

## 解决方案

利用 `feedparser-clj` 库来解析 RSS 数据。

在开始之前，请在项目依赖关系中加入 `[org.clojars.scsibug/feedparser-clj "0.4.0"]`，或用 `lein-try` 开始 REPL：

```
$ lein try org.clojars.scsibug/feedparser-clj
```

以 RSS Feed 的 URL 为参数来调用 `feedparser-clj.core/parse-feed`，取得相应的数据，并解析成 Clojure 数据：

```
(require '[feedparser-clj.core :as rss])

(rss/parse-feed (str "https://github.com/clojure-cookbook/clojure-cookbook/"
                     "commits/master.atom"))
;; -> {:authors [...]
;;     :entries [{:link "LINK" :title "TITLE" :contents "CONTENT"} ...]
;;      ...}
```

也可以用 `java.io.InputStream` 为参数调用 `parse-feed`，从文件或其他位置读取数据：

```
(with-open [writer (clojure.java.io/writer "master.atom")]
  (spit writer
        (slurp (str "https://github.com/clojure-cookbook/clojure-cookbook/"
                    "commits/master.atom"))))

(with-open [stream (clojure.java.io/input-stream "master.atom")]
  (rss/parse-feed stream))
;; -> {:authors [...]
;;     :entries [{:link "LINK" :title "TITLE" :contents "CONTENT"} ...]
;;     ...}
```

## 讨论

`feedparser-clj` 包装了 Java ROME 库，它能够处理各种格式的 RSS 和 Atom feed。`feedparser-clj.core/parse-feed` 返回一个 Clojure 映射表，非常像背后的 XML feed。

大多数时候，你关心的数据在 :entries 键中，它包含了一组映射表，对应于每个 RSS 条目。

有些 RSS feed 有 <link rel="next"> 元素，表明返回的列表不完整，依照该链接能取得更多条目。可以生成这些 RSS 条目的惰性列表：

```
(defn next-uri
  "Return the rel=next href in a feed."
  [feed]
  (-> feed
      :entry-links
      (->> (filter #(= (:rel %) "next")))
      first
      :href))


(defn lazy-stream
  "Return a lazy stream of RSS entries."
  [uri]
   (let [raw-response (rss/parse-feed uri)]
      (lazy-cat (:entries raw-response)
         (if-let [nxt (next-uri raw-response)]
            (lazy-stream nxt)))))
```

要验证惰性加载正在发生，可以在 lazy-stream 中加上日志或追踪信息，但也很容易确认取得的条目比一次性取得的条目多：

```
(def youtube-feed "http://gdata.youtube.com/feeds/api/videos")

(count (rss/parse-feed youtube-feed))
;; -> 15

(count (take 50 (lazy-stream youtube-feed)))
;; -> 50
```

在 REPL 中对惰性序列求值要谨慎，因为它会尝试打印整个序列。使用 take 以便只实现序列的一部分。

## 参阅

- 4.22 节“处理 XML 数据”，探讨了读写 RSS feed 这样的 XML 数据的更多内容。
- 5.1 节“发出 HTTP 请求”。

# 5.5 发送邮件

作者：Ryan Neufeld

## 问题

需要在 Clojure 应用中发送邮件。

## 解决方案

使用 postal 来发送邮件，它是 JavaMail 包的简单包装。

要继续这个实例，请用 lein-try 启动 REPL：

```
$ lein try com.draines/postal
```

调用 postal.core/send-message 函数来发送消息，参数是两个映射表，第一个包含连接的细节，第二个包含消息的细节。例如，通过 Gmail 账号发送一封邮件给你自己：

```
(require '[postal.core :refer [send-message]])

;; 用你自己的账户口令代替下面的
(def email "<<your gmail address>")
(def pass "<your gmail password>")

(def conn {:host "smtp.gmail.com"
           :ssl true
           :user email
           :pass pass})

(send-message conn {:from email
                    :to email
                    :subject "A message, from the past"
                    :body "Hi there, me!"})
;; -> {:error :SUCCESS, :code 0, :message "messages sent"}
```

如果一切正常，你很快就会收到来自你自己的一封邮件。

## 讨论

有了可敬的 JavaMail 作为核心，postal 就没什么可担心的了。即使是 Gmail 饱受争议的认证设置也可以用一个 :ssl 键来解决。虽然对于简单的邮件发送，我们一般会建议试试原本的 Java API，但我们更喜欢 postal，因为它的 API 面向的是数据，而不是对象。

面向数据有一点确实很棒，就是指定连接细节。send-message 函数的第一个参数是（多功能的）映射表，包含连接细节。有效的连接细节包括：

:host

SMTP 服务器的主机名。如果本地运行可以不提供。

:port

SMTP 服务器的端口。有一些根据上下文的默认值，包括在设置 :ssl 时使用 465，在设置 :tls 时使用 25。

`:user`

用于认证的用户名（如果要求认证）。

`:pass`

用于认证的口令（如果要求认证）。

`:ssl`

如果值为真，使用 SSL 加密。

`:tls`

如果值为真，使用 TLS 加密。

如果没有提供连接细节（要么省略了第一个参数，要么传入了 `nil`），postal 将试图把邮件发给本地的 sendmail 实例（http://en.wikipedia.org/wiki/Sendmail）。

> 因为亚马逊的简单邮件服务（Simple Email Service, SES）支持 SMTP，所以可以利用 `postal`，通过亚马逊的基础设施来发送邮件。

与连接细节类似，消息本身也表示为简单的数据映射表。所有的标准头都通过消息键支持：

- 发件人选项
  - `:from`
  - `:reply-to`
- 收件人选项
  - `:to`
  - `:cc`
  - `:bcc`
- 内容选项
  - `:subject`
  - `:body`
- 元数据选项
  - `:date`
  - `:message-id`
  - `:user-agent`

除了这些之外的选项将附在消息后，作为辅助的头信息。

在用 :to、:cc 或 :bcc 键来指定收件人时，值可以是单个地址，也可以是地址序列：

```
{:to "joe@example.com"
 :cc ["joe@example.com", "jim@example.com", "jeff@example.com"]
 :bcc "archive@example.com"}
```

消息体可以指定为一个字符串，或一系列部分的映射表。前面的方式发出简单的纯文本邮件，后面的方式发出多部分的 MIME 消息。MIME（多目的因特网邮件扩展）是一种标准，允许邮件包含附件或其他富文本内容，如 HTML。

部分映射表由两个值组成：:type 和 :content。对于消息体的部分，:type 是该内容的 MIME 类型，:content 是该内容的文本表示形式。例如，要创建一条消息，既包含普通文本，又包含该内容的 HTML 表示：

```
:body [:alternative
       {:type "text/plain"
        :content "You just won the lottery!"}
       {:type "text/html"
        :content "<html>
                    <body>
                      <p>You just <b>won</b> the lottery!</p>
                    </body>
                  </html>"}]
```

你会注意到，上面消息体的第一个“部分”实际上不是一个部分映射表，而是关键字 :alternative。消息通常以“混合”模式发送，告诉邮件客户端，每个部分构成了整体消息的一段。而具有 :alternative 类型的消息则告诉客户端，每个部分都表示完整的消息，但使用不同的格式。

如果需要发送复杂的多部分消息，并需要对消息的创建有更多的控制，就需要用原始的 JavaMail API 来构造消息。

对于附件，:type 参数的行为稍有不同，它控制了附件是包含在内（:inline），还是作为附件（:attachment）。通过为 :content 键提供一个 File 对象，从而指定附件的内容。附件内容的类型和名称通常从 File 对象推知，但它们可以用 :content-type 和 :file-name 键来覆写。

例如，向你所有的好朋友发送一张你的猫的照片，看起来可能是这样的：

```
:body [{:type "text/plain"
        :content "Hey folks,\n\nCheck out these pictures of my cat!"}
       {:type :inline
        :content (File. "/tmp/lester-flying-photoshop")
```

```
             :content-type "image/jpeg"
             :file-name "lester-flying.jpeg"}
            {:type :attachment
             :content (File. "/tmp/lester-upside-down.jpeg")}]
```

### 参阅

- postal 的 GitHub 代码库（https://github.com/drewr/postal）。
- JavaMail 的 API 文档（https://javamail.java.net/nonav/docs/api/）。

## 5.6　用RabbitMQ实现队列通信

作者：Ryan Neufeld，最初由 Michael Klishin 提交

### 问题

希望在一些应用之间通过外部队列来通信，比如 RabbitMQ（http://rabbitmq.com/）。

### 解决方案

Langohr（http://clojurerabbitmq.info/）是一个小 RabbitMQ 客户端，用它与 RabbitMQ 通信。

在开始之前，请在项目依赖关系中加入 `[com.novemberain/langohr "1.6.0"]`，或用 `lein-try` 开始 REPL：

```
$ lein try com.novemberain/langohr
```

为了继续这个实例，需要安装并运行 RabbitMQ（http://www.rabbitmq.com/download.html）。

安装完成后，用命令 **`rabbitmq-server`** 来启动独立的 RabbitMQ 服务器：

```
$ rabbitmq-server
```

在对 RabbitMQ 执行操作之前，必须连接到服务器，并打开通信信道。信道是中介，在它上面产生和消费消息：

```
(require 'langohr.core
         'langohr.channel)

;; 连接到本地 RabbitMQ 集群结点，localhost:5672
(def conn (langohr.core/connect {:hostname "localhost"}))

;; 在连接上打开信道
(def ch (langohr.channel/open conn))
```

在 RabbitMQ 中，消息被发布给交换器，通过绑定路由到队列，最终被消费者消费。有几

种不同类型的交换器，定义了不同的交付语义。最基本的交换器类型是直接（direct），它根据消息的路由键（routing key）来路由消息。

要在生产者和消费者之间建立管道，先调用 langohr.queue/declare，用期望的名字创建一个队列：

```
(require '[langohr.queue :as lq])

(def resize-queue "imaging.resize")

(lq/declare ch resize-queue)
;; -> {:queue "imaging.resize",
;;     :consumer-count 0,
;;     :message_count 0,
;;     :consumer_count 0,
;;     :message-count 0}
```

在默认情况下，RabbitMQ 创建空交换器（一个空字符串）和每个队列之间的绑定。现在你可以通过调用 langohr.basic/publish 向 "imaging.resize" 队列发送消息，参数是信道、直接交换、路由键（队列的名字）和消息：

```
(lb/publish ch "" resize-queue "hello.jpg")
```

要从队列中同步地消费消息，就调用 langohr.basic/get，参数是信道和队列名称：

```
(def hello-msg (lb/get ch resize-queue))

hello-msg
;; -> [{:routing-key "imaging.resize", :headers nil ...} #<byte[] [B@2b195c88>]

(String. (last hello-msg) "UTF-8")
;; -> "hello.jpg"
```

要异步地消费消息，就调用 langohr.consumers/subscribe 来订阅队列。每当有消息发布给队列时，提供给 subscribe 的处理函数将被调用：

```
(require '[langohr.consumers :as lc])

(defn resize-image-handler
  "Spawn a resize process for each resize message received"
  [ch metadata ^bytes payload]
  (let [filename (String. payload "UTF-8")]
    (println (format "Resizing file %s" filename))))

;; 订阅队列，提供处理函数
(def tag (lc/subscribe ch resize-queue resize-image-handler))

;; 订阅的返回值是一个订阅标签
tag
;; -> "amq.ctag-7hsNsSqLDEEoES5AkIC6XQ"
```

```
(lb/publish ch "" resize-queue "hello-again.jpg")
;; *out*
;; Resizing file hello-again.jpg

;; 通过标签来取消订阅
(lb/cancel ch tag)
```

# 讨论

至此，你已经了解了利用 RabbitMQ 收发消息的过程，但这仅仅是 Langohr 和 RabbitMQ 能力的一点皮毛。Langohr 是一个小型 RabbitMQ 客户端，它包装了 Java RabbitMQ 库，支持 AMQP 0-9-1 以及 RabbitMQ 对 AMQP 的扩展，并提供了一个 HTTP API 客户端。

AMQP 0-9-1 及其扩展实现 Langohr，是以几个主要概念为核心的：交换器、队列和绑定。

### 交换器

交换器非常像一个邮局：当消息发布给交换器时，交换器将它路由给一个或多个队列。这些消息路由的方式，取决于交换器的类型，以及交换器和队列之间的绑定。

有几种交换器的类型，每一种都有自己的交换语义，参见表 5-1。可以创建定制的交换类型，来处理复杂的路由场景（例如，根据内容或地理位置数据来路由），或仅仅为了方便。

表5-1　内建的交换器类型

| 名称 | 行为 | 预定义的交换 |
|---|---|---|
| Direct | 1∶1，根据路由键来路由 | `""` |
| Fanout | 1∶N，忽略路由键 | `"amq.fanout"` |
| Topic | 1∶N，考虑路由键 | `"amq.topic"` |
| Headers | 1∶1，考虑许多头部 | `"amq.match"` |

要声明一种内建的交换器，就调用 `langohr.exchange/fanout`、`langohr.exchange/topic`、`langohr.exchange/direct` 或 `langohr.exchange/headers`。每个函数都提供了对应交换器类型的相关选项，最终调用的是 `langohr.exchange/declare`：

```
(require '[langohr.exchange :as le])

;; 为图像处理完成创建 fanout 交换器
(le/fanout ch "imaging.complete")
```

交换器有一些关联属性：

- 名称；
- 类型（direct、fanout、topic、headers，或某种定制类型）；
- 持久性（它是否应该活过代理重启？）；

- 不再使用时，该交换器是否被自动删除；
- 定制的元数据（有时称为 x-arguments）。

用 langohr.exchange/declare 可以直接定制这些属性，创建自己的交换器类型。

### 队列

队列就像邮局中的邮箱。langohr.queue/declare 函数创建命名队列。除名称外，该函数还接受一些关键字参数来调整队列的属性，包括队列是否 :durable、:exclusive 或 :auto-delete。其他的参数可以通过 :arguments 值来指定。

```
(lq/declare ch "imaging.transcode" :durable true)
;; -> {:queue "imaging.transcode", ...}
```

用 langohr.queue/declare-server-named 函数可以生成具有唯一名称的队列。该函数与 langohr.queue/declare 类似，但没有名称参数：

```
(lq/declare-server-named ch)
;; -> "amq.gen-FcFv8JD9K8-4NuT8kC3jKA"
```

不像交换器，RabbitMQ 中的队列都是同样的类型。

### 绑定

正如你在解决方案中看到的，直接交换器根据名称，在默认的交换器和每个队列之间建立了一个隐含的绑定。但在自然环境下，队列通常明确地与交换器绑定。可以调用 langohr.queue/bind 来创建自己的绑定，参数是信道、队列名称和交换器名称：

```
;; 创建唯一的完成队列……
(def completion-queue (lq/declare-server-named ch))

;; 将它与 imaging.complete fanout 交换器绑定
(lq/bind ch completion-queue "imaging.complete")
```

### 发布

用 langohr.basic/publish 函数将消息发布到交换器。这个函数有以下三个主要参数（除信道外）。

- *交换器的名称*

  要么是用户生成的交换器，如 "imaging.complete"，或是内建的交换器，如 "amq.fanout" 或 ""。

- *路由键*

  由交换器使用，根据类型将消息路由给队列。

- *消息*

  发送给队列的消息体。

作为可选参数，publish 允许用户指定许多消息头部作为关键字参数。完整的清单参见 publish 函数的文档。

### 消费

声明了一些队列后，有两种方式从队列中消费消息：

- 拖拉，用 langohr.basic/get；
- 推送，用 langohr.consumers/subscribe。

在推送 API 中，可以同步调用 get 函数，从队列中取得一条消息。get 的返回值是一个元数据映射表的三元组和一个消息体。返回的消息体是一个字节数组，要得到普通文本消息，可以用字符串的构造方法（String.），将这些字节转换成字符串。因为 String 字节数组的编码是 UTF-8，所以在调用 String 构造函数时使用 "UTF-8" 编码选项，这很重要：

```
(lb/publish ch "" resize-queue "hello.jpg")
(let [[_ body] (lb/get ch resize-queue)]
  (String. body "UTF-8"))
;; -> "hello.jpg"
```

如果队列中没有消息，get 将返回 nil。

在拖拉 API 中，用 langohr.consumers/subscribe 来订阅队列，提供消息处理函数，在队列收到每条消息时调用。处理函数调用时带三个参数：信道、元数据和消息体字节数组。

```
;; 一个普通的处理函数
(defn resize-image-handler
  "Spawn a resize process for each resize message received"
  [ch metadata ^bytes payload]
  (let [filename (String. payload "UTF-8")]
    (println (format "Resizing file %s" filename))))
```

subscribe 是不阻塞的调用，结束时它会返回一个标签字符串，以后利用这个字符串来调用 langohr.consumers/cancel，实现取消订阅。

subscribe 函数也允许指定许多队列生命周期函数，在 langohr.consumers/create-default 文档中有充分的描述。

### 确认

被消费的消息需要确认。这可以自动发生（只要消息发送给消费者，RabbitMQ 就认为已确认）或手动发生。

消息被确认后，它将从队列中删除。如果在消息确认之前，信道意外关闭，RabbitMQ 会自动将它重新放入队列。请注意，这些确认具有专门针对应用的语义，有助于确保消息得到正确处理。

在手工确认时，应用负责确认或拒绝消息。这分别是通过 `langohr.basic/ack` 和 `langohr.basic/nack` 完成的，每个函数都接受一个元数据属性，名为 `delivery-tag`（发送 ID）。要启用手工确认，在调用 `langohr.consumers/subscribe` 时传入 `:auto-ack false`：

```
(defn manual-ack-handler
  "Spawn a resize process for each resize message received"
  [ch {:keys [delivery-tag]} ^bytes payload]
  (try
    (String. payload "UTF-8")
    ;; 做一些工作，然后确认该消息
    (lb/ack ch delivery-tag)
    (catch Throwable t
      ;; 拒绝该消息
      (lb/nack ch delivery-tag))))

(lc/subscribe ch resize-queue manual-ack-handler :auto-ack false)
```

请注意，如果将消息退回队列，而队列又只有一个消费者，它会马上重新发送。

在收到一条确认之前，也可以控制向客户端推送多少消息。这被称为*预先取设置*，通过调用 `langohr.basic/qos` 实现。这个设置适用于整个信道：

```
;; 预先取一打消息
(lb/qos ch 12)
```

RabbitMQ 队列也可以在集群节点之间镜像，实现高可用性，限定消息的长度或超时等。更多的信息，参见 RabbitMQ 和 Langohr 的文档网站。

## 参阅

- Langohr 文档（http://clojurerabbitmq.info/）。
- Langohr 的 API 参考（http://reference.clojurerabbitmq.info/）。
- RabbitMQ 教程（http://rabbitmq.com/getstarted.html）。
- 如果需要低层次的 RabbitMQ 访问，也许可以研究一下利用 Clojure 的 Java 互操作性，使用 RabbitMQ 的 Java 客户端（http://www.rabbitmq.com/java-client.html），Langohr 正是基于这个库。

# 5.7 通过MQTT与嵌入式设备通信

作者：Sandeep Nangia

## 问题

希望用发布 / 订阅的方式，与嵌入式设备通信（想想“物联网”）。

## 解决方案

使用 Machine Head（https://github.com/clojurewerkz/machine_head），它是一个 Clojure 库，通过 MQTT（http://mqtt.org/）协议实现机器到机器（M2M）的通信。该协议要求有一个 MQTT 代理，所有设备（或机器）通过它来通信，即发布或订阅特定主题的消息。可以使用 Mosquitto（http://mosquitto.org/）代理和它的测试用安装实例，地址是 tcp://test.mosquitto.org:1883（当然，你的机器需要有可用的互联网连接）。

要继续这个实例，请用 `lein-try` 启动 REPL：

```
$ lein try clojurewerkz/machine_head
```

开始创建一个简单的 `connect-and-subscribe` 函数，监听一个主题，并打印它收到的消息：

```
(require '[clojurewerkz.machine-head.client :as mh])

(defn message-handler [topic meta payload]
  (let [p (apply str (map char payload))]
    (println "received " p "on topic " topic)))

(defn connect-and-subscribe [broker-addr topics subscriberid]
  (let [qos-levels (vec (repeat (count topics) 2)) ;; All at qos 2
        conn-sub (mh/connect broker-addr subscriberid)]
    (if (mh/connected? conn-sub)
      (do
        (mh/subscribe conn-sub topics message-handler {:qos qos-levels})
        conn-sub)))) ;; Return conn-sub for later mh/disconnect...

(def subscriberid (mh/generate-id))
;; or use a unique id
;; (def subscriberid "SNSubscriber01")

(connect-and-subscribe "tcp://test.mosquitto.org:1883"
                       ["SNControlNetwork/Florida/device1"] subscriberid)
```

打开另一个终端窗口，启动第二个 `lein-try` REPL 会话。利用下面的代码向代理发布消息。请注意，订阅者必须已经连接，这样才不会丢失传来的消息：

```
(require '[clojurewerkz.machine-head.client :as mh])

(defn connect-and-publish [broker-addr client-id topic]
  (let [qos 2
        retained false
        conn (mh/connect broker-addr client-id)]
    (if (mh/connected? conn)
      (do (dotimes [n 5]
            (let [payload (str "msg" n)]
              (mh/publish conn topic payload qos retained)
              (println "published " payload)))
          (mh/disconnect conn)))))
```

```
(def pubclientid (mh/generate-id))
pubclientid
;; -> "ryan.1384135173618"

(connect-and-publish "tcp://test.mosquitto.org:1883" pubclientid
                       "SNControlNetwork/Florida/device1")
;; *out* of publish REPL
;; published  msg0
;; published  msg1
;; published  msg2
;; published  msg3
;; published  msg4
;; *out* of client REPL
;; received  msg0 on topic SNControlNetwork/Florida/device1
;; received  msg1 on topic SNControlNetwork/Florida/device1
;; received  msg2 on topic SNControlNetwork/Florida/device1
;; received  msg3 on topic SNControlNetwork/Florida/device1
;; received  msg4 on topic SNControlNetwork/Florida/device1
```

## 讨论

MQTT（http://mqtt.org/）是开放的、轻量级的发布 / 订阅消息协议。它适用于带宽很珍贵或连接不可靠的情况。AMQP 协议在商业消息的各种场景下有优势，但 MQTT 通常是较小负载和最后一公里的选择，因为它容易在硬件中实现。MQTT 协议具有以下的属性，使得它适用于受限制的网络。

- 针对资源有限的设备而设计，例如电池驱动的 8 位控制器。
- 内部压缩成以位计的头和可变长的字段。包的最小可能大小只有 2 字节。
- 不需要轮询。实现异步双向消息推送。
- 支持常连接和有时连接两种模式。
- 在低带宽网络中测试过，如 VSAT 和 GPRS。

该协议定义了 3 种可能的服务质量（QoS）值：`0`、`1` 和 `2`，对应于发送后不管、至少一次和刚好一次的服务质量。QoS 参数 `1` 和 `2` 要求客户端有持久存储，这样才能保存消息，直到收到确认。在上面的实例中，使用的库提供了默认的持久实现。

MQTT 也有消息保留的概念。如果在 `connect-and-publish` 函数中将 `retained` 设置为 `true`，代理将记住该主题最后一条保留消息。在订阅者连接时，代理将发给它最后一条消息（该消息的 `retained` 为 `true`），不必等待接收下一条消息。

> WebSphere 和 RabbitMQ（http://www.rabbitmq.com/mqtt.html）也实现了 MQTT，可以替代 Mosquitto。虽然前面的代码使用的是 Mosquitto 测试代理（tcp://test.mosquitto.org:1883），但也可以安装自己的 Mosquitto 代理，参阅 MQTT 安装说明（https://github.com/mqtt/mqtt.github.io/wiki/doku.php/mosquitto_message_broker）。

主题常常用分隔符 / 来定义层级关系。例如，某个领域 SNControl 的传感器设备可能将它们的值发布到 SNControl/Florida/device1、SNControl/Florida/device2 等主题。同时，领域 RKNControl 的设备可能将它们的值发布到 RKNControl/Washington/device1 等主题。以这种方式命名主题有助于利用通配符订阅多个主题。

下面是通配符的用法。

- /

  用作分隔符。

- +

  单层通配符，可以出现在字符串的任何地方。

- #

  多层通配符，需要出现在字符串的末尾。

例如，可能有下面的订阅。

- SNControl/#

  在 SNControl/Florida 下的所有设备（如 SNControl/Florida/device1/sensor1 和 SNControl/Florida/device1/sensor2）以及 SNControl/California/device1 将匹配。

- SNControl/+/device1

  所有州中在 SNControl 领域下的 device1 将匹配（如 SNControl/Florida/device1 和 SNControl/California/device1）。

- SNControl/+/+/sensor1

  所有州中在 SNControl 下的 sensor1 将匹配（如 SNControl/Florida/device1/sensor1 和 SNControl/Florida/device2/sensor1）。

在前面的代码中，connect-and-subscribe 方法使用了回调处理函数 message-handler 来处理从代理接收到的消息。在 connect-and-subscribe 方法中，Machine Head 库的 connect 方法调用时，提供了代理的地址和客户端 ID（通过 generate-id 生成，或其他某种唯一 ID）。然后它用 connected? 方法检查连接是否已经建立。调用 subscribe 方法时，参数是连接、包含订阅主题的向量、消息处理函数和 :qos 选项。然后订阅者等待一段时间，用 disconnect 方法断开连接。

connect-and-publish 方法调用了 connect 方法，后者接受代理地址和客户端 ID 作为参数，返回连接 conn。然后它用 connected? 方法检查连接是否成功，并调用 publish 方法向代理发布消息（发了几次）。publish 方法接受的参数是连接、主题字符串、消息体、QoS 值和 retained。QoS 值 2 对应于“刚好一次”发送。retained 值为 false 告诉代理不要保留消

息。最后，`disconnect` 方法断开与代理的连接。

虽然前面的代码片段只是打印出收到的消息，但你可以用别的方式来使用这些消息（例如，根据代码收到的警报触发某些动作）。

### 参阅

- MQTT 协议网站（http://mqtt.org/）。
- Machine Head 库（https://github.com/clojurewerkz/machine_head）的文档（http://clojuremqtt.info/）。
- Eclipse Paho 库（http://www.eclipse.org/paho/），Machine Head 底层使用这个 Java 库实现 MQTT 通信。
- Mosquitto（http://mosquitto.org/），实现了 MQTT 协议的开源消息代理。
- *Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry*（http://www.redbooks.ibm.com/abstracts/sg248054.html, IBM 红皮书），作者 Valerie Lampkin 等，该书更详细地解释了 MQTT。
- Andy Stanford-Clark 的 TED 演讲（http://www.youtube.com/watch?v=s9nrm8q5eGg），他是 MQTT 发明者之一，该演讲风格幽默而信息丰富，探讨了如何使用 MQTT。

## 5.8 并发使用ZeroMQ

作者：Kevin J. Lynagh

### 问题

希望并发地使用 ZeroMQ，但 ZeroMQ 套接字对线程来说不是安全的。可以利用锁或其他 Java 并发原语手工建立互斥访问，但你可能希望使用更简单的方法。

### 解决方案

用 `zmq-async`（https://github.com/lynaghk/zmq-async）库，利用 `core.async` 来简化 ZeroMQ 的并发使用。

为了继续本实例，你的系统应该安装 ZeroMQ 3.2。如果系统是 Mac 并安装了 Homebrew（http://brew.sh/）包管理器，用下面的命令来安装它：

```
$ brew install zeromq
```

或者，在 Ubuntu 系统上：

```
$ apt-get install libzmq3
```

或者，访问 ØMQ 的下载页面（http://zeromq.org/intro:get-the-software）。

在开始之前，请在项目依赖关系中加入 `[com.keminglabs/zmq-async "0.1.0"]`，或用 `lein-try` 开始 REPL：

```
$ lein try com.keminglabs/zmq-async
```

下面是用 `core.async` 中的两个异步 `go` 语句块进行简单的来回通信，通过 ZeroMQ 的进程内套接字进行。

```
(require '[com.keminglabs.zmq-async.core :refer [register-socket!]]
         '[clojure.core.async :refer [>! <! go chan sliding-buffer close!]])

(def addr "inproc://ping-pong")

(def server-in  (chan (sliding-buffer 64)))
(def server-out (chan (sliding-buffer 64)))
(def client-in  (chan (sliding-buffer 64)))
(def client-out (chan (sliding-buffer 64)))

(register-socket! {:in server-in
                   :out server-out
                   :socket-type :rep
                   :configurator (fn [socket] (.bind socket addr))})

(register-socket! {:in client-in
                   :out client-out
                   :socket-type :req
                   :configurator (fn [socket] (.connect socket addr))})

(do
  ;; 一个简单的服务器工作者，等待进来的请求，回应以 "pong"
  (go
    (dotimes [_ 3]
      (println (String. (<! server-out)))
      (>! server-in "pong"))
    (close! server-in))

  ;; 一个简单的客户端工作者，发出 "ping" 请求，等待回应
  (go
    (dotimes [_ 3]
      (>! client-in "ping")
      (println (String. (<! client-out))))
    (close! client-in)))
;; *out*
;; ping
;; pong
;; ping
;; pong
;; ping
;; pong
```

## 讨论

ZeroMQ 是面向消息的套接字系统，在多种传输层（进程内、进程间、机器间通过 TCP 等）上支持多种通信方式（请求 / 应答、发布 / 订阅等），与多种语言绑定。ZeroMQ 套接字提供了很好的基础，来构建面向服务的架构。ZeroMQ 的套接字比 HTTP 的开销小，在架构上更灵活，除了请求 / 应答之外，还支持发布 / 订阅、扇出和其他拓扑结构。

但是，ZeroMQ 套接字对线程来说不是安全的，并发使用通常需要明确加锁，或专门的线程和队列。`zmq-async` 库解决了这些问题，替你创建了 ZeroMQ 套接字，让你通过线程安全的 `core.async` 信道来访问它们。

`zmq-async` 库提供了一个函数 `com.keminglabs.zmq-async.core/register-socket!`，它将 ZeroMQ 套接字与一个或两个 `core.async` 信道关联起来：`:in`（可以向它写入字符串或字节数组）和 `:out`（可以从它读取字节数组）。用 `>!` 写入 Clojure 字符串集合或字节数组时，发出一条多部分消息。收到的多部分消息放在 `core.async` 信道中。用 `<!` 读取这些消息，将得到一个包含字节数组的向量。

为了模拟两个异步进程通过 ZeroMQ 交互，前面的例子用了两个 `go` 语句块，通过注册的信道读写消息。每个 `go` 语句块都会在后台线程中立即开始执行。“服务器”语句块将等待并应答三次请求（`<!` 会阻塞，直到它接收到值），每次都应答“pong”。同时，“客户端”语句块将发起三次“ping”请求，每次都会等到应答后再发出下次请求。最后，在两个语句块完成了工作之后，各自用 `close!` 关闭了信道。

`register-socket!` 函数可以接受一个已经创建的 ZeroMQ 套接字，但通常你会通过传入 `:socket-type` 和 `:configurator`，利用该库创建一个套接字。`configurator` 是一个函数，接受原始的 ZeroMQ 套接字对象。该函数在套接字创建之后运行，以便连接 / 绑定地址，设置 pub/sub 订阅，或配置该套接字。

> 支持 `register-socket!` 的隐含上下文一次只能处理一条进或出的消息。如果你需要多个套接字并行工作（例如，你不希望因为在另一个套接字上读取 10 GB 的消息，而错失一条小小的控制消息），那就需要多个 `zmq-async` 上下文。

## 参阅

- Rich Hickey 的“Language of the System”演讲（http://www.youtube.com/watch?v=ROor6_NGIWU,），其中他概括了队列的好处。
- ZeroMQ 指南（http://zguide.zeromq.org/）介绍了架构模式和建议。
- 3.11 节“用 `core.async` 解除消费者和生产者的耦合”。

- `core.async` 的介绍性博客文章（http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html），提供了很好的概述。

# 5.9 创建TCP客户端

作者：Luke VanderHart

## 问题

希望与远程主机的特定端口建立 TCP 连接。

## 解决方案

利用 Java 互操作性创建 `java.net.Socket` 实例，连接到远程主机。

例如，下面的代码利用 `Socket` 创建一个 TCP 连接，并发送 HTTP GET 请求，将结果以字符串的形式返回：

```
(require '[clojure.java.io :as io])
(import '[java.io StringWriter]
        '[java.net Socket])

(defn send-request
  "Sends an HTTP GET request to the specified host, port, and path"
  [host port path]
  (with-open [sock (Socket. host port)
              writer (io/writer sock)
              reader (io/reader sock)
              response (StringWriter.)]
    (.append writer (str "GET " path "\n"))
    (.flush writer)
    (io/copy reader response)
    (str response)))
```

这个函数取得 `java.io.Writer` 和 `java.io.Reader` 实例，向远程服务器发送数据，并从服务器接收数据。通过拼接满足 HTTP 规范的字符串并写入 writer 对象，它形成了一个最基本的 HTTP 客户端，向指定的服务端口发出 GET 请求。然后利用工具函数 `clojure.java.io/copy`，将结果复制到 `java.io.StringWriter` 实例，并作为字符串返回。

在 REPL 中调用 `(send-request "google.com" 80 "/")` 将返回一个非常长的字符串，包含访问 Google 主页时的整个 HTTP 响应。

## 讨论

这个例子利用 `clojure.java.io` 命名空间得到了 `java.io.Writer` 和 `java.io.Reader` 实例，

实现通过网络套接字读写文本数据。事实上，套接字实例并非真的对文本数据有所限制，而且也可以通过 `clojure.java.io/input-stream` 和 `clojure.java.io/output-stream`，很容易地获取原始的二进制输入输出流。但因为 HTTP 是一个文本协议，所以利用 `Reader` 和 `Writer` 的高层特性更有意义。

> 这个例子使用 HTTP 是因为它是很多读者熟悉的协议。在真实世界中，使用原始的 TCP 套接字来发出 HTTP 请求几乎肯定是糟糕的想法。有些库提供了更高层的接口，实现了 HTTP 的请求和应答，封装了许多麻烦的细节，诸如转义（escaping）、编码和格式等。

还要注意 reader、writer 和套接字本身都在 `with-open` 宏的上下文之中。这确保了它们的工作结束时，会调用 `close` 方法，释放 TCP 连接。如果连接不释放，它将继续消耗客户端和服务器端的资源，可能需要远端来终止它。

如果在 `with-open` 上下文中返回惰性序列，要用 `doall` 来完全实现这些序列，这一点很重要。这是因为 `with-open` 打开的资源仅在 `with-open` 语句块中可用。`doall` 函数完全实现一个集合，返回它在内存中的全部内容：

```
(realized? (range 100))
;; -> false

(realized? (doall (range 100)))
;; -> true
```

根据具体的应用，你可能倾向于用 `doseq` 宏。除了保持整个序列外，`doseq` 对序列中的每个元素执行它的语句体。如果需要对序列中的每个元素产生副作用，这就很有用，但需要坚持做完所有的事情：

```
(doseq [n (range 3)]
  (println n))
;; *out*
;; 0
;; 1
;; 2
```

## 参阅

- 5.10 节“创建 TCP 服务器”。
- TCP 协议的维基百科（http://en.wikipedia.org/wiki/Transmission_Control_Protocol）。

# 5.10 创建TCP服务器

作者：Luke VanderHart

## 问题

希望在一个端口打开套接字，作为低级 TCP 服务器。

## 解决方案

利用 Java 的互操作性和 `java.net.ServerSocket` 类，创建 TCP 监听者。利用 `clojure.java.io` 中的函数来获取输入输出流（或 reader 和 writer 对象），通过套接字读写数据：

```
(require '[clojure.java.io :as io])
(import '[java.net ServerSocket])

(defn receive
  "Read a line of textual data from the given socket"
  [socket]
  (.readLine (io/reader socket)))

(defn send
  "Send the given string message out over the given socket"
  [socket msg]
  (let [writer (io/writer socket)]
      (.write writer msg)
      (.flush writer)))

(defn serve [port handler]
  (with-open [server-sock (ServerSocket. port)
              sock (.accept server-sock)]
    (let [msg-in (receive sock)
          msg-out (handler msg-in)]
      (send sock msg-out))))
```

这段代码定义了三个函数。`receive` 和 `send` 负责利用套接字读取和写入字符串数据，用到了 `clojure.java.io/reader` 和 `clojure.java.io/writer` 函数。它们都接受 `java.net.Socket` 作为参数，返回从套接字的输入和输出流构造的 `java.io.Reader` 和 `java.io.Writer`。

`server` 负责在特定端口创建 `ServerSocket` 实例。它也接受一个处理函数，该函数将用于处理进入的请求，决定响应的消息。

在创建了 `ServerSocket` 实例后，`server` 马上调用它的 `accept` 方法，它将阻塞，直到 TCP 连接建立。当客户端连接时，它以 `java.net.Socket` 实例的形式返回该会话。

然后它将套接字传递给 `receive` 函数，该函数在套接字上打开 reader 对象并阻塞，直到接收到一行完整的输入，以换行符（`\n`）结束。在它接收到一行时，它会用收到的值来调用处理函数，利用在同一个套接字上打开的 writer 对象，调用 `send` 来发送响应。`send` 也调用 writer 对象的 `flush` 方法，确保所有数据真正发送回客户端，而不是放在 `Writer` 实例的缓存中。

在发送完响应后，`serve` 方法返回。因为它在创建服务器套接字和 TCP 会话套接字时用了 `with-open` 宏，所以在返回之前，它会对这些套接字调用 `close` 方法，断开客户端并终止会话。

如果要尝试一下，请在 REPL 中调用 `serve` 函数。一个简单的例子，就用 `(serve 8888 #(.toUpperCase %))`。注意，它不会马上返回，而是会阻塞，等待客户端连接。

要连接服务器，可以使用 telnet 客户端，几乎在所有操作系统上，它都是默认安装的。要用它，就打开命令行窗口：

```
$ telnet localhost 8888
Trying ::1...
Connected to localhost.
Escape character is '^]'.
```

这时你可以输入任何内容（在下面的例子中，输入是“Hello, World!”）。在完成输入后，确保按下回车键来发送换行字符：

```
$ telnet localhost 8888
Trying ::1...
Connected to localhost.
Escape character is '^]'.
Hello, World!
HELLO, WORLD!Connection closed by foreign host
```

你会看到，当你输入换行时，服务器的响应是你的输入的大写版本（由于处理函数的原因），然后马上终止了连接。在 REPL 中，你会看到 `serve` 函数终于返回了。

# 讨论

这个例子使用了 reader 和 writer 对象，它们只处理文本数据，这是为了让使用套接字的概念更容易展示。当然，实际的套接字不限于字符串，可以发送和接收任何类型的二进制数据。

要做到这一点，只要使用 `clojure.java.io/input-stream` 和 `clojure.java.io/output-stream` 函数，而不是 `clojure.java.io/reader` 和 `clojure.java.io/writer` 函数。它们将返回 `java.io.InputStream` 和 `java.io.OutputStream` 对象。这些对象提供了读写原始字节的 API，而不只是读取字符串和字符。

关于这个例子，你可能注意到了一件事，即它不像传统的服务器，在 `serve` 函数返回后，它不再继续接受连接请求。要持续使用，通常你会希望它能服务多个连接请求。

幸运的是，有了 Clojure 提供的并发工具，这是比较容易的。将 `serve` 函数改成持续工作的服务器，需要三处改动：

- 在独立的线程中运行服务器，这样就不会阻塞 REPL；
- 在处理完第一个请求后，不要关闭服务器套接字；
- 在处理完请求后，马上循环处理下一个请求。

而且，由于服务器将运行在 REPL 之外的线程中，所以好的做法是提供一种机制来终止服务器，而不是终止整个 JVM。修改后的代码是这样的：

```
(defn serve-persistent [port handler]
  (let [running (atom true)]
    (future
      (with-open [server-sock (ServerSocket. port)]
        (while @running
          (with-open [sock (.accept server-sock)]
            (let [msg-in (receive sock)
                  msg-out (handler msg-in)]
              (send sock msg-out))))))
    running))
```

这段代码的主要特点是，它在一个 future 语句块中异步启动了服务器套接字，并在一个循环中调用 `accept` 方法。它也创建了一个名为 `running` 的原子类型，并返回它，在每次循环中都检查它。要停止服务器，只要将它设为 `false`，循环就会中止：

```
(def a (serve-persistent 8888 #(.toUpperCase %)))
;; -> #'my-server/a

;; 服务器在运行，将响应多个请求

(reset! a false)
;; -> false
;; 服务器停止了,不再响应下一个请求
```

## 何时使用套接字

在这些例子中可以看到，原始的服务器套接字是相当低层的网络构造。要有效使用它们就意味着要么创建自己的数据协议，要么重新实现已有的数据协议，而且要自己处理所有费时累人的连接、清空缓存、断开输入输出流等工作。

如果你的通信需要满足已有的协议或通信技术（如 HTTP、SSH 或消息队列），那么几乎肯定应该直接使用它们。这些协议有许多可用的服务器和库，可以在较高的抽象层次上编程，具有更好的性能和弹性。

但是，理解这些不同技术的底层工作原理仍然是有价值的。至少在 JVM 中，大多数网络代码最终都归结为调用本实例描述的原始套接字机制。要理解高级网络工具（如 HTTP 请求或 JMS 队列）的工作原理，先理解套接字的工作原理是很关键的。

## 参阅

- Java 的 `ServerSocket` 和 `Socket` 对象的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/net/ServerSocket.html）。
- `clojure.java.io` 命名空间的 API 文档（http://clojure.github.io/clojure/clojure.java.io-api.html）。
- 5.9 节“创建 TCP 客户端”。
- 关于 TCP 协议的维基百科（http://en.wikipedia.org/wiki/Transmission_Control_Protocol）。

# 5.11 收发UDP包

作者：Luke VanderHart

## 问题

应用需要发送或接收异步的 UDP 包。

## 解决方案

利用 Java 互操作性以及 `java.net.DatagramSocket` 和 `java.net.DatagramPacket` 类，发送和接收 UDP 消息。

下面的例子展示了实现收发短字符串的函数，这些字符串被编码成 UDP 包：

```
(import '[java.net DatagramSocket
                   DatagramPacket
                   InetSocketAddress])

(defn send
  "Send a short textual message over a DatagramSocket to the specified
  host and port. If the string is over 512 bytes long, it will be
  truncated."
  [^DatagramSocket socket msg host port]
  (let [payload (.getBytes msg)
        length (min (alength payload) 512)
        address (InetSocketAddress. host port)
        packet (DatagramPacket. payload length address)]
    (.send socket packet)))

(defn receive
  "Block until a UDP message is received on the given DatagramSocket, and
  return the payload message as a string."
  [^DatagramSocket socket]
  (let [buffer (byte-array 512)
        packet (DatagramPacket. buffer 512)]
    (.receive socket packet)
    (String. (.getData packet)
```

```
               0 (.getLength packet))))

(defn receive-loop
  "Given a function and DatagramSocket, will (in another thread) wait
  for the socket to receive a message, and whenever it does, will call
  the provided function on the incoming message."
  [socket f]
  (future (while true (f (receive socket)))))
```

send 函数相当简单，因为它的大部分内容都是构造一个字节数组，作为 DatagramPacket 的负载，并调用构造方法。最有趣的是负载大小限制为 512 字节，利用了 DatagramPacket 构造方法的 length 参数。因为在一个 UDP 包中发送超过 512 字节的负载，通常不安全。虽然某些网络基础设施可能支持，但另一些并不支持。

receive 函数创建了接收数据的字节数组，将它添加到可修改的空 DatagramPacket 实例中，并对套接字调用 DatagramSocket.receive 方法。在收到数据时，receive 方法将填充 DatagramPacket 实例并返回。然后这段 Clojure 代码利用填充的字节数组的范围（即 0 到 DatagramPacket.getLength 方法返回的值），构造并返回新的 String。

因为 receive 函数会阻塞，而且只返回一个值，所以在接受多条消息或在 REPL 中使用时，它不是特别有用。receive-loop 包装了 receive 函数，在一个独立的线程中反复调用它。如果它返回值，就调用提供的函数，然后循环等待更多的输入。

要执行这段代码，首先要创建 DatagramSocket 实例，在 REPL 中：

```
(def socket (DatagramSocket. 8888))
;; -> #'udp/socket
```

在指定的端口（这里是 8888）创建 UDP 套接字。

接下来，用 receive-loop 函数启动一个监听者。对于这个例子，只要传入 println 函数，就能打印出所有收到的值：

```
(receive-loop socket println)
;; -> #<core$future_call$reify__6267@2783890e: :pending>
```

然后就可以发送消息了！如果用 receive-loop 启动监听线程无误，应该马上就能看到打印出收到的消息：

```
(send socket "hello, world!" "localhost" 8888)
;; *out*
;; hello, world!
;;
;; -> nil
```

这个例子是发送到 localhost，消息传递非常快，以至于 send 函数还没返回，消息就已经收到了。

## 讨论

不像 TCP，UDP（用户数据报协议）是一个异步协议，不保证消息到达的次序，内容是否正确，甚至不保证消息是否到达。与 TCP 这样的协议相比，在交换机中，UDP 通常有较低的包延迟时间，因为不需要执行错误检查和恢复。

在决定使用 UDP 之前，要确保应用设计能够在包丢失或出错的情况下也能继续工作。

由于 UDP 使用异步消息作为模型，所以很容易用 `core.async` 来包装原始的 `DatagramSocket` 实例。`core.async` 提供了很好的信道抽象，让你能够以清晰的、管理良好的方式，消费和产生本质上异步的事件（如 UDP 消息）。

## UDP组播

利用一种名为 UDP 组播的技术，UDP 也可以将同一个数据包发往多个目的地。要使用组播，需要创建 `java.net.MulticastSocket` 实例来替代 `java.net.DatagramSocket` 实例。

Oracle 网站上的文档（http://docs.oracle.com/javase/7/docs/api/java/net/MulticastSocket.html）很好地解释了如何使用 `MulticastSocket`，此处不再赘述，因为就是直接利用 Java 的互操作性。看过前面的例子，扩展到 `MulticastSocket` 应该不言自明。

## 参阅

- 3.11 节“用 `core.async` 解除消费者和生产者的耦合”。
- 5.9 节“创建 TCP 客户端”。
- 5.10 节“创建 TCP 服务器”。
- `java.net.MulticastSocket` 的 API 文档（http://docs.oracle.com/javase/7/docs/api/java/net/MulticastSocket.html）。

# 第6章 数据库

## 6.0 简介

在当今时代，将数据存储在数据库中是开发者经常遇到的任务，这是理所当然的事情。和世界上大多数的编程语言一样，Clojure 也有一群驱动和客户端，与数据库交互。但 Clojure 的不同之处在于，它能够组合。

本书在前面曾提到：在 Clojure 中，数据为王。你会发现许多数据库客户端库，它们干一点点跑腿的活，然后马上为你让开道路。这些库这样做不是因为懒（至少我们希望如此），而是出于关注点分离的原则：我来处理数据库连接，你来处理领域知识（数据）。实际上，最好的 API 是基于数据库，只提供一两个函数，让你操作数据和查询，这些数据直接以 Clojure 数据结构的方式插入。

在本章中，我们将探讨许多数据库和技术，包括 SQL、全文本查找、Mongo、Redis 和 Datomic。

Datomic（http://www.datomic.com/）是数据库领域中比较有趣的新进展。它由 Rich Hickey（你可能知道，就是此人写出了 Clojure）发明和维护，是可伸缩的、支持事务的、面向值的、考虑时间的数据库，其遵循的原则和原理与 Clojure 相同。如果你喜欢 Clojure，就肯定应该试试 Datomic，它既可以作为应用的数据存储库，也可以作为学习工具，进一步探索函数式、面向数据的编程。

# 6.1 连接SQL数据库

作者：Tom Hicks，最初由 Simone Mosciatti 提交

## 问题

程序需要连接 SQL 数据库。

## 解决方案

用 `clojure.java.jdbc` 库，通过 JDBC 的方式访问 SQL 数据库。

要继续这个实例，就需要连接一个运行的 SQL 数据库和表。我们建议用 PostgreSQL[1]。

在 PostgreSQL 运行后（假定运行在 localhost:5432），执行下面的命令，创建这个实例需要的数据库：

```
# 在 Mac 中：
$ /Applications/Postgres93.app/Contents/MacOS/bin/createdb cookbook_experiments

# 在其他环境中：
$ createdb cookbook_experiments
```

在开始之前，请在项目依赖关系中加入 `[org.clojure/java.jdbc "0.3.0"]`。你也需要选定的 RDBMS 的 JDBC 驱动。如果要继续这个实例，就用 `[org.postgresql/postgresql "9.2-1003-jdbc4"]`。要用 `lein-try` 开始 REPL，可输入下面的 Leiningen 命令：

```
$ lein try org.clojure/java.jdbc "0.3.0" \
           java-jdbc/dsl "0.1.0" \
           org.postgresql/postgresql "9.2-1003-jdbc4"
```

要用 `clojure.java.jdbc` 与数据库交互，只需要一个连接规格说明。这个规格说明采用普通的 Clojure 映射表的形式，其中的值表明了数据库驱动类型、位置和认证信息：

```
(def db-spec {:classname "org.postgresql.Driver"
              :subprotocol "postgresql"
              :subname "//localhost:5432/cookbook_experiments"
              ;; 对不保密的本地数据库不需要……
              ;; :user "bilbo"
              ;; :password "secret"
              })
```

调用 `clojure.java.jdbc/create-table` 函数，带上规格说明和不定数目的列规格说明，就能在指定的数据库中创建一个关系表：

注 1：Mac 用户：访问 http://postgresapp.com/，下载易于安装的 DMG。其他人：可以在 PostgreSQL wiki（https://wiki.postgresql.org/wiki/Detailed_installation_guides）找到相应操作系统的安装指南。

```
(require '[clojure.java.jdbc :as jdbc]
         '[java-jdbc.ddl :as ddl])

(jdbc/db-do-commands db-spec false
  (ddl/create-table
     :tags
     [:id :serial "PRIMARY KEY"]
     [:name :varchar "NOT NULL"]))
;; -> (0)
```

许多其他函数查询和操作数据库，如 `clojure.java.jdbc/insert!`，它们以数据库规格说明作为第一个参数：

```
(require '[java-jdbc.sql :as sql])

(jdbc/insert! db-spec :tags
                      {:name "Clojure"}
                      {:name "Java"})
;; -> ({:name "Clojure", :id 1} {:name "Java", :id 2})

(jdbc/query db-spec (sql/select * :tags (sql/where {:name "Clojure"})))
;; -> ({:name "Clojure", `:id 1})
```

## 讨论

`clojure.java.jdbc` 库提供了一些函数，包装了 Java JDBC 规范的基本功能。此外，来自 `java-jdbc/dsl` 项目的 `java-jdbc.sql` 和 `java-jdbc.ddl` 命名空间，实现了小型的 DSL，能生成基本的 SQL DML 和 DDL 语句。

因为 `clojure.java.jdbc` 库基于 Java JDBC，所以它可以用于许多最流行的 SQL 数据库，包括 Apache Derby、HSQLDB、Microsoft SQL Server、MySQL、PostgreSQL 和 SQLite。

建立和访问数据源所需的参数被称为*数据库规格说明*（常常简写为 db-spec），在一个简单的 Clojure 映射表中提供。这个规格说明中的参数通常包括驱动程序类名、特定 RDBMS 类型的子协议、主机名、端口号、数据库名，以及用户名和口令。

`clojure.java.jdbc` 库也允许其他几种数据源规格说明，包括 Java URI、已经打开的连接，JNDI 连接和普通字符串。例如，可以用 `:connection-uri` 键提供完整的 URI 字符串：

```
;; 作为规格说明字符串
(def db-spec
  "jdbc:postgresql://bilbo:secret@localhost:5432/cookbook_experiment")

;; 作为连接 URI 映射表……
;; 包括用户名和口令……
(def db-spec
  {:connection-uri (str "jdbc:postgresql://localhost:5432/cookbook_experiments?"
                        "user=bilbo&password=secret")})
```

```
;; 或者不包括
(def db-spec
  {:connection-uri "jdbc:postgresql://localhost:5432/cookbook_experiments"})
```

数据库记录表示为 Clojure 映射表，表的列名作为键。取出一组数据库记录就得到一个映射表序列，然后可以用所有普通的 Clojure 函数处理：

```
(jdbc/query db-spec (sql/select * :tags))
;; -> ({:name "Clojure", :id 1}
;      {:name "Java", :id 2})

(filter #(not (.endsWith (:name %) "ure"))
       (jdbc/query db-spec (sql/select * :tags)))
;; -> ({:name "Java", :id 2})
```

还有其他一些 Clojure 库可以访问关系数据库，每个都提供了不同的抽象和 DSL，用于操作 SQL 数据和表达式，比如 Korma。但是，`clojure.java.jdbc` 包含了日常数据库访问的大部分需求。

## 参阅

- 6.2 节“利用连接池连接 SQL 数据库”，学习利用 BoneCP 和 `clojure.java.jdbc` 实现 SQL 数据库连接池。
- 6.3 节“操作 SQL 数据库”，学习利用 `clojure.java.jdbc` 与 SQL 数据库交互。
- 访问 `clojure.java.jdbc` 的 GitHub 代码库（https://github.com/clojure/java.jdbc），了解该库的更多信息。
- 访问 `java-jdbc/dsl` 的 GitHub 代码库（https://github.com/seancorfield/jsql），了解关于它的 SQL 查询生成功能。或者，研究 Honey SQL（https://github.com/jkk/honeysql）、SQLingvo（https://github.com/r0man/sqlingvo），或 Korma 库（http://sqlkorma.com/），了解 SQL 查询生成。Korma 将在 6.4 节中讨论。

# 6.2 利用连接池连接SQL数据库

作者：Tom Hicks 和 Filippo Diotalevi

## 问题

希望利用连接池高效地连接 SQL 数据库。

## 解决方案

使用 BoneCP 连接和语句池库来包装基于 JDBC 的驱动，创建数据资源池。然后，池中的数据资源被 `clojure.java.jdbc` 库使用，正如 6.1 节“连接 SQL 数据库”中描述的一样。

要继续这个实例，就需要连接一个运行的 SQL 数据库和表。我们建议用 PostgreSQL[2]。

在 PostgreSQL 运行后（假定运行在 localhost:5432），执行下面的命令，创建这个实例需要的数据库：

```
# 在 Mac 中：
$ /Applications/Postgres93.app/Contents/MacOS/bin/createdb cookbook_experiments

# 在其他环境中：
$ createdb cookbook_experiments
```

在开始之前，请在项目依赖关系中加入 BoneCP（`[com.jolbox/bonecp "0.8.0.RELEASE"]`），还需要加入适合 RDBMS 的 JDBC 库。你还需要 SLF4J 日志库。或者，可以用 `lein-try` 开始 REPL：

```
$ lein try com.jolbox/bonecp "0.8.0.RELEASE" \
           org.clojure/java.jdbc "0.3.0" \
           java-jdbc/dsl "0.1.0" \
           org.postgresql/postgresql "9.2-1003-jdbc4" \
           org.slf4j/slf4j-nop # Just do not log anything
```

首先，创建一个数据库规格说明，其中包含访问数据库的参数。它包含的键说明了池的最初规模和最大规模，以及分区数。

```
(def db-spec {:classname "org.postgresql.Driver"
              :subprotocol "postgresql"
              :subname "//localhost:5432/cookbook_experiments"
              :init-pool-size 4
              :max-pool-size 20
              :partitions 2})
```

要创建放入池中的 `BoneCPDataSource` 对象，就要定义一个函数（为了方便）来利用数据库规格映射表中的参数：

```
(import 'com.jolbox.bonecp.BoneCPDataSource)

(defn pooled-datasource [db-spec]
  (let [{:keys [classname subprotocol subname user password
                init-pool-size max-pool-size idle-time partitions]} db-spec
        min-connections (inc (int (/ init-pool-size partitions)))
        max-connections (inc (int (/ max-pool-size partitions)))
        cpds (doto (BoneCPDataSource.)
                   (.setDriverClass classname)
                   (.setJdbcUrl (str "jdbc:" subprotocol ":" subname))
                   (.setUsername user)
                   (.setPassword password)
```

注 2：Mac 用户：访问 http://postgresapp.com/，下载易于安装的 DMG。其他人：可以在 PostgreSQL wiki（https://wiki.postgresql.org/wiki/Detailed_installation_guides）找到相应操作系统的安装指南。

```
                    (.setMinConnectionsPerPartition min-connections)
                    (.setMaxConnectionsPerPartition max-connections)
                    (.setPartitionCount partitions)
                    (.setStatisticsEnabled true)
                    (.setIdleMaxAgeInMinutes (or idle-time 60)))]
        {:datasource cpds}))
```

利用这个方便的函数来定义一个放入池中的数据源，连接到数据库：

```
(def pooled-db-spec (pooled-datasource db-spec))

pooled-db-spec
;; -> {:datasource #<BoneCPDataSource ...>}
```

将这个数据库规格作为所有 `clojure.java.jdbc` 函数的第一个参数，来查询或操作数据库：

```
(require '[clojure.java.jdbc :as jdbc]
         '[java-jdbc.ddl :as ddl]
         '[java-jdbc.sql :as sql])

(jdbc/db-do-commands pooled-db-spec false
  (ddl/create-table
    :blog_posts
    [:id :serial "PRIMARY KEY"]
    [:title "varchar(255)" "NOT NULL"]
    [:body :text]))
;; -> (0)

(jdbc/insert! pooled-db-spec
              :blog_posts
              {:title "My first post!" :body "This is going to be good!"})
;; -> ({:body "This is going to be good!", :title "My first post!", :id 1})

(jdbc/query pooled-db-spec
            (sql/select * :blog_posts (sql/where{:title "My first post!"})))
;; -> ({:body "This is going to be good!", :title "My first post!", :id 1})
```

## 讨论

正如解决方案中展示的，`clojure.java.jdbc` 库可以利用 JDBC 数据源创建数据库连接，这样 BoneCP 或其他连接池库就很容易将连接放入池中。

BoneCP 库包装了原有的 JDBC 类，允许创建高效的数据源。它可以适配传统的无池驱动和数据源，将它们扩展为 `Connection` 和 `PreparedStatement` 实例池。

虽然该库提供了几种创建数据源的方式，但大多数用户会发现这里提供的例子是最容易的。

BoneCP 提供了几十个配置参数，控制对数据源及其连接的操作。幸运的是，大多数配置参数都有默认值。可以通过参数指定来控制各个方面，如连接池的最小规模、最大规模和

初始规模、空闲连接数、连接的时间、事务处理、`PreparedStatement` 池的使用，以及是否、何时和如何测试池中的连接。

通过调用库中 `BoneCPDataSource` 类的 `close` 方法，可以释放池中的数据资源（线程和数据库连接）。关闭后再试图使用池中的数据源，将导致 SQL 异常：

```
(.close (:datasource pooled-db-spec))
;; -> nil
```

## 参阅

- 6.1 节“连接 SQL 数据库”，了解用 `clojure.java.jdbc` 实现基本数据库连接。
- 6.3 节“操作 SQL 数据库”，学习利用 `clojure.java.jdbc` 与 SQL 数据库交互。
- BoneCP 的 文 档（http://jolbox.com/index.html?page=http://jolbox.com/configuration.html）和 GitHub 代码库（https://github.com/wwadge/bonecp）。
- `clojure.java.jdbc` 的 GitHub 代码库（https://github.com/clojure/java.jdbc），了解该库的更多信息。

# 6.3　操作 SQL 数据库

作者：Tom Hicks

## 问题

Clojure 程序需要操作 SQL 数据库中的表和记录。

## 解决方案

用 `clojure.java.jdbc` 库，基于 JDBC 来访问 SQL 数据库。

要继续这个实例，就需要连接一个运行的 SQL 数据库和表。我们建议用 PostgreSQL[3]。

在 PostgreSQL 运行后（假定运行在 localhost:5432），执行下面的命令，创建这个实例需要的数据库：

```
# 在 Mac 中 :
$ /Applications/Postgres93.app/Contents/MacOS/bin/createdb cookbook_experiments

# 在其他环境中 :
$ createdb cookbook_experiments
```

注 3：Mac 用户：访问 http://postgresapp.com/，下载易于安装的 DMG。其他人：可以在 PostgreSQL wiki（https://wiki.postgresql.org/wiki/Detailed_installation_guides）找到相应操作系统的安装指南。

在开始之前，请在项目依赖关系中加入 `[org.clojure/java.jdbc "0.3.0"]` 和 `[java-jdbc/dsl"0.1.0"]`。还需选定 RDBMS 的 JDBC 驱动。如果要继续这个实例，请用 `[org.postgresql/post gresql "9.2-1003-jdbc4"]`。要用 `lein-try` 开始 REPL，请输入以下 Leiningen 命令：

```
$ lein try org.clojure/java.jdbc "0.3.0" \
           java-jdbc/dsl "0.1.0" \
           org.postgresql/postgresql "9.2-1003-jdbc4"
```

然后定义如何访问该数据库：

```
(def db-spec {:classname "org.postgresql.Driver"
              :subprotocol "postgresql"
              :subname "//localhost:5432/cookbook_experiments"})
```

要创建新表，请用 `java-jdbc.ddl/create-table` 函数来生成必要的 DDL 语句，然后将该语句传递给 `jdbc/db-do-commands` 函数执行：

```
(require '[clojure.java.jdbc :as jdbc]
         '[java-jdbc.ddl :as ddl])

(jdbc/db-do-commands db-spec
  (ddl/create-table :fruit
    [:name "varchar(16)" "PRIMARY KEY"]
    [:appearance "varchar(32)"]
    [:cost :int "NOT NULL"]
    [:unit "varchar(16)"]
    [:grade :real]))
;; -> (0)
```

利用 `clojure.java.jdbc/insert!` 函数，向表中插入完整的记录。调用它时，每行表示为一个向量，其中包含各列的值。要确保提供的列值与表中声明的列顺序一致：

```
(jdbc/insert! db-spec :fruit
  nil ; 列名省略了
  ["Red Delicious" "dark red" 20 "bushel" 8.2]
  ["Plantain" "mild spotting" 48 "stalk" 7.4]
  ["Kiwifruit" "fresh" 35 "crate" 9.1]
  ["Plum" "ripe" 12 "carton" 8.4])
;; -> (1 1 1 1)
```

要查询数据库，就用 `java-jdbc.sql/select` 函数生成查询的 SQL，然后调用 `clojure.java.jdbc/query` 来执行：

```
(require '[java-jdbc.sql :as sql])

(jdbc/query db-spec
  (sql/select * :fruit (sql/where {:appearance "ripe"})))
;; -> ({:grade 8.4, :unit "carton", :cost 12, :appearance "ripe", :name "Plum"})
```

如果不再需要某个表，就用 java-jdbc.ddl/drop-table 生成相应的 DDL 语句，再调用 clojure.java.dbc/jdb-do-commands 来执行：

```
(jdbc/db-do-commands db-spec
  (ddl/create-table :delete_me
    [:name "varchar(16)" "PRIMARY KEY"]))

(jdbc/db-do-commands db-spec (ddl/drop-table :delete_me))
;; -> (0)
```

# 讨论

clojure.java.jdbc 库提供了一些函数，包装了 Java JDBC 规范的基本功能。java-jdbc/dsl 项目的 java-jdbc.sql 和 java-jdbc.ddl 命名空间，实现了小型的 DSL，能生成基本的 SQL DML 和 DDL 语句。

> java-jdbc/dsl 曾经是 clojure.java.jdbc 的一部分，但后来被移除，目的是让核心库的 API 尽可能小。

java-dbc.ddl/create-table 函数生成了创建表所需的 DDL。参数是一个表名和一个向量，向量中包含每列的规格说明。在本书编写时，表级的规格说明还不支持。

### 插入和更新记录

记录可以用多种方式插入表中。除了前面提到的向量方式，clojure.java.jdbc/insert! 函数还可以接受一个或多个映射表，其中以列名作为键：

```
(jdbc/insert! db-spec :fruit
  {:name "Banana" :appearance "spotting" :cost 35}
  {:name "Tomato" :appearance "rotten" :cost 10 :grade 1.4}
  {:name "Peach" :appearance "fresh" :cost 37 :unit "pallet"})
;; -> ({:grade nil, :unit nil, :cost 35, :appearance "spotting", :name "Banana"}
;;     {:grade 1.4, :unit nil, :cost 10, :appearance "rotten", :name "Tomato"}
;;     {:grade nil, :unit "pallet", :cost 37, :appearance "fresh",
;;      :name "Peach"})
```

如果插入一行而不指定某些列的值，可以在调用 clojure.java.jdbc/insert! 时以包含列名的向量作为第一个参数，后面是一个或多个向量，包含对应列的值：

```
(jdbc/insert! db-spec :fruit
  [:name :cost]
  ["Mango" 84]
  ["Kumquat" 77])
;; -> (1 1)
```

要更新已有的记录，就调用 clojure.java.jdbc/update!，参数是一个映射表，包含列名和对应的新值。可选的 java-jdbc.sql/where 子句控制需要更新哪些行：

```
(jdbc/update! db-spec :fruit
  {:grade 7.0 :appearance "spotting" :cost 75}
  (sql/where {:name "Mango"}))
;; -> (1)
```

## 事务

提供数据库事务是为了确保多个操作以原子方式执行（即都执行或都不执行）。clojure.java.jdbc/with-db-transaction 宏利用数据库规格说明，创建了感知事务的连接。使用这个感知事务的连接来实现事务：

```
;; 原子地插入两种新水果
(jdbc/with-db-transaction [trans-conn db-spec]
  (jdbc/insert! trans-conn :fruit {:name "Fig" :cost 12})
  (jdbc/insert! trans-conn :fruit {:name "Date" :cost 14}))
;; -> ({:grade nil, :unit nil, :cost 14, :appearance nil, :name "Date"})
```

如果抛出异常，事务将回滚：

```
;; 查询表中现在有多少记录
(defn fruit-count
  "Query how many items are in the fruit table."
  [db-spec]
  (let [result (jdbc/query db-spec (sql/select "count(*)" :fruit))]
    (:count (first result))))

(fruit-count db-spec)
;; -> 11

(jdbc/with-db-transaction [trans-conn db-spec]
  (jdbc/insert! trans-conn :fruit
    [:name :cost]
    ["Grape" 86]
    ["Pear" 86])
  ;; 这时 insert! 调用已经完成，但事务还没完成
  ;; 异常将导致事务回滚，所以数据库没有变化
  (throw (Exception. "sql-test-exception")))
;; -> Exception sql-test-exception ...

;; 表的记录数仍然一样
(fruit-count db-spec)
;; -> 11
```

事务可以通过 clojure.java.jdbc/db-set-rollback-only! 函数显式地设置为回滚。这种设置可以通过 clojure.java.jdbc/db-unset-rollback-only! 函数取消，通过 clojure.java.jdbc/is-rollback-only 函数来测试：

```
(fruit-count db-spec)
;; -> 11

(jdbc/with-db-transaction [trans-conn db-spec]
  (jdbc/db-set-rollback-only! trans-conn)
  (jdbc/insert! trans-conn :fruit {:name "Pear" :cost 69}))
;; -> ({:grade nil, :unit nil, :cost 69, :appearance nil, :name "Pear"})

;; 表的记录数仍然一样
(fruit-count db-spec)
;; -> 11
```

### 读取和处理记录

查询返回的数据库记录是 Clojure 映射表，表的列名用作映射表中的键。取回一组数据库记录就得到一系列的映射表，能由所有的普通 Clojure 函数处理。下面，我们查询 fruit 表中的全部记录，收集所有低品质水果的名称和等级：

```
(->> (jdbc/query db-spec (sql/select "name, grade" :fruit))
     ;; 过滤所有水果，找出 grade < 3.0 的
     (filter (fn [{:keys [grade]}] (and grade (< grade 3.0))))
     (map (juxt :name :grade)))
;; -> (["Tomato" 1.4])
```

前面例子中使用了 `java-jdbc.sql` 命名空间提供的 SQL DSL。这个 DSL 实现是 SQL 语句生成的简单抽象。目前，它提供了一些基本的机制，支持 select、join、`where` 子句和 `order-by` 子句：

```
(defn fresh-fruit []
  (jdbc/query db-spec
    (sql/select [:f.name] {:fruit :f}
      (sql/where {:f.appearance "fresh"})
      (sql/order-by :f.name))))

(fresh-fruit)
;; -> ({:name "Kiwifruit"} {:name "Peach"})
```

使用 SQL DSL 完全是可选的。要更直接地控制，可以向 `query` 函数传递一个向量，其中包含 SQL 查询字符串和参数。下面的函数也找出了低品质的水果，但它直接将品质阈值传递给了 SQL 语句：

```
(defn find-low-quality [acceptable]
  (jdbc/query db-spec
              ["select name, grade from fruit where grade < ?" acceptable]))

(find-low-quality 3.0)
;; -> ({:grade 1.4, :name "Tomato"})
```

`jdbc/query` 函数有一些可选的关键字参数，控制它如何构造返回的结果集。`:result-set-fn` 参数指定了一个函数，它将作用于整个结果集（一个惰性序列），然后再返回。默认的

参数是 doall 函数：

```
(defn hi-lo [rs] [(first rs) (last rs)])

;; 找出价格最高和最低的水果
(jdbc/query db-spec
            ["select * from fruit order by cost desc"]
            :result-set-fn hi-lo)

;; -> [{:grade nil, :unit nil, :cost 77, :appearance nil, :name "Kumquat"}
;;     {:grade 1.4, :unit nil, :cost 10, :appearance "rotten", :name "Tomato"}]
```

:row-fn 参数指定了一个函数，在结果集生成时，它将应用于结果集中的每一行。默认的参数是 identity 函数：

```
(defn add-tax [row] (assoc row :tax (* 0.08 (row :cost))))

(jdbc/query db-spec
             ["select name,cost from fruit where cost = 12"]
             :row-fn add-tax)
;; -> ({:tax 0.96, :cost 12, :name "Plum"} {:tax 0.96, :cost 12, :name "Fig"})
```

Boolean 类型的参数 :as-arrays? 表明返回的结果集是否作为一组向量。默认的参数值是 false：

```
(jdbc/query db-spec
            ["select name,cost,grade from fruit where appearance = 'spotting'"]
            :as-arrays? true)
;; -> ([:name :cost :grade] ["Banana" 35 nil] ["Mango" 75 7.0])
```

最后，:identifiers 参数指定了一个函数，它将作用于结果集中的每个列名。默认的参数是 clojure.string/lower-case 函数，它取表中列名的小写形式，然后再转为关键字。如果应用需要对列名进行某种不同的转换，就通过这个关键字参数提供另一个函数。

对于快速简便地访问大多数流行的关系型数据库，clojure.java.jdbc 库是一个好选择。它使用 Clojure 的向量和映射表来表示记录，这与 Clojure 强调面向数据的编程是一致的。SQL 的初学者可以利用提供的 DSL，而专业用户可以直接构造并执行复杂的 SQL 语句。

## 参阅

- 6.1 节“连接 SQL 数据库”，了解用 clojure.java.jdbc 实现基本数据库连接。
- 6.2 节“利用连接池连接 SQL 数据库”，学习利用 BoneCP 和 clojure.java.jdbc 实现 SQL 数据库连接池。
- clojure.java.jdbc 的 GitHub 代码库（https://github.com/clojure/java.jdbc），了解该库的更多信息。

- 访问 `java-jdbc/dsl` 的 GitHub 代码库（https://github.com/seancorfield/jsql），了解关于它的 SQL 查询生成功能。或者，研究 Honey SQL（https://github.com/jkk/honeysql）、SQLingvo（https://github.com/r0man/sqlingvo），或 Korma 库（http://sqlkorma.com/），了解 SQL 查询生成。Korma 将在 6.4 节讨论。

## 6.4　用Korma简化SQL

作者：Dmitri Sotnikov 和 Chris Allen

### 问题

希望处理存储在关系数据库中的数据，又不用手写 SQL。

### 解决方案

使用 Korma 作为 DSL 来生成 SQL 查询，并遍历关系。

在开始之前，请在项目依赖关系中加入 `[korma "0.3.0-RC6"]` 和 `[org.postgresql/postgresql "9.2-1002-jdbc4"]`，或用 `lein-try` 开始 REPL：

```
$ lein try korma org.postgresql/postgresql
```

要继续这个实例，就需要连接一个运行的 SQL 数据库和表。我们建议用 PostgreSQL[4]。

在 PostgreSQL 运行后（假定运行在 localhost:5432），执行下面的命令，创建这个实例需要的数据库：

```
# 在 Mac 中：
$ /Applications/Postgres.app/Contents/MacOS/bin/createdb learn_korma

# 在其他环境中：
$ createdb learn_korma
```

要连接 `learn_korma` 数据库，就用 `defdb` 和 `postgres` 辅助函数。因为 Korma 是相当大的 DSL，所以可以接受将它的内容 `:refer :all` 到模型命名空间中：

```
(require '[korma.db :refer :all])

(defdb db
  (postgres {:db "learn_korma"}))
```

注 4：Mac 用户：访问 http://postgresapp.com/，下载易于安装的 DMG。其他人：可以在 PostgreSQL wiki（https://wiki.postgresql.org/wiki/Detailed_installation_guides）找到相应操作系统的安装指南。

要与数据库中的表进行交互，先定义并创建 Korma 所谓的实体。下面将定义博客文章的实体：

```
(defentity posts
  (pk :id)
  (table :posts) ; 表名
  (entity-fields :title :content)) ; 默认要SELECT的字段
```

一般会利用合适的迁移库（migration library）作为 schema，但为了简单，我们手工创建一个表。用 exec-raw 函数对数据库执行原始的 SQL 语句。只有在非常有必要时才这样做：

```
(def create-posts (str "CREATE TABLE posts "
                       "(id serial, title text, content text,"
                       "created_on timestamp default current_timestamp);"))

(exec-raw create-posts)
```

既然 posts 表已经存在，就可以对 posts 调用 insert，并带上值的映射表作为参数，向数据库中添加记录。每条记录都表示为一个映射表。映射表中键的名称必须与数据库的列名匹配：

```
(insert posts
        (values nil {:title "First post" :content "blah blah blah"}))
```

要从数据库中取得值，就用 select 来查询。成功的查询将返回一个映射表序列，每个映射表中包含的键表示列名：

```
(select posts (limit 1))
;; -> [{:created_on #inst "2013-11-01T19:21:10.652920000-00:00",
;;      :content "blah blah blah",
;;      :title "First post",
;;      :id 1}]
```

要纠正或改变原有的记录，就用 update 宏。对 posts 调用 update，提供 set-fields 声明来指定应该改变什么，提供 where 声明来缩小要改变的记录的范围：

```
(update posts
        (set-fields {:title "Best Post"})
        (where {:title "First post"}))
;; -> {:title "Best Post", :id 1 ...}
```

delete 宏与 update 类似，但不接受 set-fields 声明：

```
(delete posts
        (where {:title "Best Post"}))

(select posts)
;; -> []
```

## 讨论

Korma 提供了一种简单而直观的方式，从 Clojure 中构建 SQL 查询。使用 Korma 的好处在于，查询是用正常的代码写成，而不是 SQL 字符串。你可以很容易地编写查询，抽象出公共的操作。

Korma 通过它的实体系统提供了这些能力。实体是在传统 SQL 表之上的抽象，遮盖了 SQL 令人不快的复杂和麻烦的 DDL（数据定义语言）。通过 `defentity` 宏，可以获得传统 SQL 的所有功能，这些功能被包装成可读的、基于 Clojure 的 DSL。

在用 `defentity` 定义实体时，可以传入一些选项。常用的选项包括指定表名的 `table`，指定默认 ID 字段的 `pk`（主键），指定 SELECT 语句默认字段的 `entity-fields`，甚至还有指定实体属于哪个数据库的 `db`。

实体也简化了表间关系的定义。实体声明语句，如 `has-one`、`has-many`、`belongs-to` 和 `many-to-many` 等，定义了它与其他实体的关系。请考虑为每篇博客文章添加一个作者：

```
;; 创建 authors，假定 posts 有 author_id 字段
(defentity authors
  ;; 默认情况下，外键是 :authors_id，但这有点别扭
  (has-many posts {:fk :author_id}))

;; 重新定义 posts，它假定有 author_id 字段
(defentity posts
  (belongs-to authors {:fk :author_id}))

;; 创建 authors 表
(exec-raw "CREATE TABLE authors (id serial, name text);")

;; 在 posts 中添加 authors_id 字段
(exec-raw "ALTER TABLE posts ADD COLUMN author_id int;")

(def ryan (insert authors (values {:name "Ryan"})))
ryan
;; -> {:name "Ryan", :id 1}

(insert posts (values [{:title "My first post!", :author_id (:id ryan)}
                       {:title "My second post.",:author_id (:id ryan)}]))
(select posts
        (where {:author_id (:id ryan)}))
;; -> [{:author_id 1,
;;       ...
;;       :title "My first post!",
;;       :id 4}
;;      {:author_id 1,
;;       ...
;;       :title "My second post.",
;;       :id 5}]
```

基于它的实体系统，Korma 提供了常见 SQL 语句的 DSL 版本，如 `select`、`update`、`insert` 和 `delete`。最有趣的查询类型是 `select`，它几乎支持 SELECT 语句的所有选项，包括简化的表连接（通过它的关系辅助函数）。一些值得注意的辅助函数包括 `aggregate`、`join`、`order`、`group` 和 `having`。很可能 SQL 语句的每个功能，在 Korma 中都有对应的辅助函数。

Korma 的 DSL 不仅方便，而且可组合。使用 `select*` 而不是 `select`，将查询返回为一个值，而不是执行后的结果。你可以连接多个查询值，通过常规的 `select` 辅助函数来建立或保存部分查询。最后，对查询值调用 `select`，执行它并得到结果：

```
(defn authors-posts
  "Retrieve all posts for a person with a given name"
  [name]
  (-> (select* posts)
      (with authors)
      (where {:authors.name name})))

;; 找到作者名为 "Ryan" 的所有文章
(-> (authors-posts "Ryan")
    (where (like :title "%second%"))
    (fields :title)
    select)
;; -> [{:title "My second post."}]
```

Korma 提供的另一种方便是默认连接。你可能已经注意到，在例子中我们从未引用我们定义的 `db`。如果只定义了一个连接，它将作为默认值，不需要明确地传递。如果乐意，也可以定义多个连接，将一系列语句包装在 `with-db` 调用中：

```
(with-db db
  (select (authors-posts "Ryan")))
```

## 参阅

- Korma 项目官方页面（http://sqlkorma.com/docs）。

# 6.5 用Lucene进行全文查找

作者：Osbert Feng

## 问题

希望在无结构或半结构化的数据集中，用 Lucene 实现灵活的全文查找。例如，你想找到在美国的、所有职位描述中包含“Clojure”的人。

## 解决方案

使用 Clucy（https://github.com/weavejester/clucy），它是 Lucene 的 Clojure 包装。Clucy 提供了一些工具，在 Clojure 进程中构建和查询索引。

要继续这个实例，需要创建一个新项目（**`lein new text-search`**），在项目依赖关系中添加 `[clucy "0.4.0"]`，并用 **`lein repl`** 启动 REPL[5]。

下面的代码创建并查询了简单的内存索引：

```
(require '[clucy.core :as clucy])

(def index (clucy/memory-index))
;; -> #'user/index


(clucy/add index
   {:name "Alice" :description "Clojure expert"
    :location "North Carolina, United States"}
   {:name "Bob" :description "Clojure novice"
    :location "Berlin, Germany"}
   {:name "Eve" :description "Eavesdropper"
    :location "Maryland, United States"})
;; -> nil

(clucy/search index "description:clojure AND location:\"united states\"" 10)
;; -> ({:name "Alice",
;;       :location "North Carolina, United States",
;;       :description "Clojure expert"})
```

## 讨论

Lucene 是一个信息检索的 Java 库。要使用 Lucene，需要生成文档并对它们进行索引，以便将来检索。文档由字段和词语构成。这个例子中的文档相当小，但 Lucene 也能高效地索引大量的巨型文档。

Clucy 包装了 Lucene，方便在 Clojure 中使用，并能够直接从简单的 Clojure 映射表生成 Lucene 文档。映射表中的键对应于字段，值对应于要索引的文本数据。

`clucy.core/search` 接受索引、查询字符串和返回结果的条数作为参数。Lucene 能够高效地查询部分是因为，它不需要返回所有匹配的文档，只要返回前 `n` 个最佳匹配。

> Clucy 不能很好地支持映射表中嵌套的值。要正确地索引和检索，请确保将值转换成简单的字符串。

注 5：我们一般建议使用 `lein-try`，但这个插件目前与 Clucy 不兼容。

这个例子使用了 memory-index，它将索引存在系统内存中。在多数真正的应用中，你会希望将索引持久在磁盘上，这样索引就能超出可用内存的大小，并且你可以重新启动进程而不必重新建立索引。Clucy 通过 disk-index 函数支持创建 Lucene 磁盘索引：

```
(def index (clucy.core/disk-index "/tmp/index"))
```

作为文档生成过程的一部分，Lucene 对字符串调用一个分析器，生成用于索引的词语。默认的 StandardAnalyzer 适用于多数情况，并能够定制一个“排除词”（stop word）列表，在词语生成时忽略：

```
(import 'org.apache.lucene.analysis.standard.StandardAnalyzer)
;; -> org.apache.lucene.analysis.standard.StandardAnalyzer

(import 'org.apache.lucene.analysis.util.CharArraySet)
;; -> org.apache.lucene.analysis.util.CharArraySet

(def stop-words
  (doto (CharArray. clucy.core/*version* 3 true)
    (.add "do")
    (.add "not")
    (.add "index")))

(binding [clucy.core/*analyzer* (StandardAnalyzer.
                                 clucy.core/*version*
                                 stop-words)]
  ;; 在这里调用索引添加和查找，在 binding 之内
  )
```

但在另一些情况下，可能需要使用不同的分析器，或自己写的分析器。例如，EnglishAnalyzer 利用了波特词干（Porter stemming）和其他技术，更适合考虑复数和所有格的情况：

```
(import org.apache.lucene.analysis.en.EnglishAnalyzer)
;; -> org.apache.lucene.analysis.en.EnglishAnalyzer

(binding [clucy.core/*analyzer* (EnglishAnalyzer. clucy.core/*version*)]
  ;; 在这里调用索引添加和查找形式，在 binding 之内
  )
```

基本的查询语法是 field:term。默认情况下，多个子句表示 OR 查找，所以如果要求子句同时为真，就要明确使用 AND。

如果不指定字段，就会使用隐含的字段 _content，将对映射表中所有的值进行索引。

返回的文档由 Lucene 默认的相关性算法来排序，该算法考虑了词频、距离和文档长度：

```
(clucy.core/search index "clojure united states" 10)
;; -> ({:name "Alice",
;;      :location "North Carolina, United States",
```

```
;;       :description "Clojure expert"}
;;      {:name "Eve",
;;       :location "Maryland, United States",
;;       :description "Eavesdropper"}
;;      {:name "Bob",
;;       :location "Berlin, Germany",
;;       :description "Clojure novice"})
```

## 参阅

- Lucene 项目主页（http://lucene.apache.org/）。
- Clucy 的 GitHub 代码库（https://github.com/weavejester/clucy）。

# 6.6 用ElasticSearch建立数据索引

作者：Michael Klishin

## 问题

希望用 ElasticSearch（http://elasticsearch.org/）索引和查找引擎，来建立数据索引。

## 解决方案

使用 Elastisch（http://clojureelasticsearch.info/），它是 ElasticSearch Java API 的最小 Clojure 包装。

为了成功执行本节中的实例，你应该在本地系统中安装并运行 ElasticSearch。ElasticSearch 网站（http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/setup.html）上有详细的安装指南。

ElasticSearch 支持多种传输方式（例如 HTTP、原生的、基于 Netty 的传输和 Memcached）。Elastisch 支持 HTTP 和原生传输。本节在实例中使用 HTTP 传输客户端，并在讨论小节中解释了如何换成原生传输。

要继续这个实例，请在项目依赖关系中添加 `[clojurewerkz/elastisch "1.2.0"]`，或用 `lein-try` 启动 REPL：

```
$ lein try clojurewerkz/elastisch
```

在用 Elastisch 建立索引和查找之前，需要告诉 Elastisch 使用什么 ElasticSearch 节点。要使用 HTTP 传输，就用 `clojurewerkz.elastisch.rest/connect!` 函数，它的唯一参数是连接端点：

```
(require '[clojurewerkz.elastisch.rest :as esr])

(esr/connect! "http://127.0.0.1:9200")
```

### 索引

在数据能被查找之前，需要建立索引。建立索引的过程是扫描文本，建立查找词语的列表，以及名为*查找索引*的数据结构。查找索引让 ElasticSearch 这样的查找引擎能够高效地取得查询的相关文档。

索引过程包括以下几步。

(1) 创建索引。
(2) [ 可选 ] 定义映射（文档应该如何被索引）。
(3) 通过 HTTP 或其他 API 提交要索引的文档。

要创建索引，就用 `clojurewerkz.elastisch.rest.index/create` 函数：

```
(require '[clojurewerkz.elastisch.rest.index :as esi])

(esr/connect! "http://127.0.0.1:9200")

;; 用给定设置创建索引，没有定制的映射类型
(esi/create "test1")

;; 创建带有定制设置的索引
(esi/create "test2" :settings {"number_of_shards" 1}))
```

完整解释可用的索引设置超出了本实例的范围。更多细节请参考 Elastisch 关于索引的文档（http://clojureelasticsearch.info/articles/indexing.html）。

### 创建映射

映射定义了文档中的字段，以及每个字段有哪些索引特点。在创建索引时，用 `:mapping` 选项指定映射类型：

```
(esr/connect! "http://127.0.0.1:9200")

;; 映射类型映射表结构与 ElasticSearch API 参考中的一样
(def mapping-types {"person"
                    {:properties {:username {:type "string" :store "yes"}
                                  :first-name {:type "string" :store "yes"}
                                  :last-name {:type "string"}
                                  :age       {:type "integer"}
                                  :title     {:type "string"
                                              :analyzer "snowball"}
                                  :planet    {:type "string"}
                                  :biography {:type "string"
                                              :analyzer "snowball"
```

```
                                              :term_vector
                                              "with_positions_offsets"}}}})

(esi/create "test3" :mappings mapping-types)))
```

### 索引文档

要将文档添加到索引，就用 clojurewerkz.elastisch.rest.document/create 函数。这将使文档 ID 自动生成：

```
(require '[clojurewerkz.elastisch.rest.document :as esd])

(esr/connect! "http://127.0.0.1:9200")

(def mapping-types {"person"
                    {:properties {:username  {:type "string" :store "yes"}
                                  :first-name {:type "string" :store "yes"}
                                  :last-name  {:type "string"}
                                  :age        {:type "integer"}
                                  :title      {:type "string" :analyzer "snowball"}
                                  :planet     {:type "string"}
                                  :biography  {:type "string"
                                               :analyzer "snowball"
                                               :term_vector
                                               "with_positions_offsets"}}}})

(esi/create "test4" :mappings mapping-types)

(def doc {:username "happyjoe"
          :first-name "Joe"
          :last-name "Smith"
          :age 30
          :title "The Boss"
          :planet "Earth"
          :biography "N/A"})


(esd/create "test4" "person" doc)
;; => {:ok true, :_index people, :_type person,
;;     :_id "2vr8sP-LTRWhSKOxyWOi_Q", :_version 1}
```

clojurewerkz.elastisch.rest.document/put 将文档添加到索引中，但希望提供一个文档 ID。

```
(esr/put "test4" "person" "happyjoe" doc)
```

## 讨论

当文档被添加到 ElasticSearch 索引中时，它先被分析。

分析过程包括以下几个步骤。

- 分词（将字段的值分解为词，即 token）。
- 过滤或修改词。
- 将分词与字段名组合，得到词语（term）。

文档究竟如何分析，决定了怎样的查询将匹配（找到）它。ElasticSearch 基于 Apache Lucene（http://lucene.apache.org/），它为开发者提供了几种分析器，实现他们需要的某种查询品质或性能。例如，不同的语言需要不同的分析器：英文、中文、阿拉伯文和俄文不能用同样的方式分析。

可以跳过对字段的分析，并指定字段的值是否保存在索引中。没有保存的字段仍将被查找，但不会包含在查找结果中。

ElasticSearch 允许用户定义不同类型的文档如何索引、分析和保存。

ElasticSearch 对*多客户组织管理*的支持很好。ElasticSearch 集群实际上可以有无数种索引和映射类型。例如，在 SaaS（软件作为服务）产品中，你可以让每个账户或组织机构使用一个独立的索引。

用 ElasticSearch 来索引文档有两种方式：提交要索引的文档时可以没有 ID，或者提供 ID 来更新文档，在这种情况下，如果文档已存在，它将被更新（会创建一个新版本）。

虽然在开发的早期使用自动创建的索引很好也很常见，但手工创建索引让你能够完成很多配置，指定 ElasticSearch 如何索引数据，从而确定可以对数据执行哪些查询。数据如何索引主要由*映射表*控制。它们定义了文档中的哪些字段需要索引，是否需要分析和如何分析，是否被保存。ElasticSearch 中的每个索引都有一个或多个*映射类型*。映射类型可以看成是数据库中的表（尽管这种类比并非总是正确）。在 ElasticSearch 中，映射类型是索引的核心，提供了对许多 ElasticSearch 功能的访问。

例如，博客应用程序可能有 article、comment 和 person 这样的类型。每个类型有不同的*映射设置*，定义了属于该类型的一组字段文档，它们应该如何索引（从而确定对它们可以进行怎样的查询），每个字段的语言是什么，等等。在应用程序中正确设置映射类型，是好的查找体验的关键。

映射类型定义了文档字段和它们的核心类型（例如字符串、整数或日期 / 时间）。设置以 JSON 文档的形式提供给 ElasticSearch，ElasticSearch 网站（http://www.elasticsearch.org/guide/reference/mapping/）提供了设置的文档。

利用 Elastisch，映射设置被指定为结构相同（schema）的 Clojure 映射表。下面是一个小例子：

```
{"tweet" {:properties {:username   {:type "string" :index "not_analyzed"}}}}
```

关于映射设置可以定义哪些东西，下面是一个简短的、很不完整的列表。

- 文档字段及其类型，它们是否需要分析。
- 文档的生存时间（TTL）。
- 文档类型是否被索引。
- 特殊字段（"_all"，默认字段等）。
- 文档层面的增强（http://www.elasticsearch.org/guide/reference/mapping/boost-field.html）。
- 时间戳字段（http://www.elasticsearch.org/guide/reference/mapping/timestamp-field.html）。

如果用 `clojurewerkz.elastisch.rest.index/create` 函数创建索引，映射设置就通过 `:mappings` 选项传入，像前面看到的那样。如果需要更新索引的映射，可以用 `clojurewerkz.elastisch.rest.index/update-mapping` 函数：

```
(esi/update-mapping "myapp_development" "person"
                    :mapping {:properties
                              {:first-name {:type "string" :store "no"}}})
```

在映射配置中，设置被作为映射表传入，其中键是名称（字符串或关键字），值是实际设置的映射表。在下面的例子中，唯一的设置是 `:properties`，它定义了一个字段——一个不需要分析的字符串：

```
{"tweet" {:properties {:username   {:type "string" :index "not_analyzed"}}}}
```

关于索引和映射选项还有很多内容，但超出了本实例的范围。详尽的功能清单请参考 Elastisch 的索引文档（http://clojureelasticsearch.info/articles/indexing.html）。

## 参阅

- 官方 ElasticSearch 指南（http://www.elasticsearch.org/guide/）。
- Elastisch 的主页（http://clojureelasticsearch.info/）。

# 6.7　使用Cassandra

作者：Alex Petrov

## 问题

希望使用 Cassandra 中存储的数据。

## 解决方案

用 Cassaforte 库（http://clojurecassandra.info/）来连接 Cassandra 集群，处理数据库中的记录。

为了成功执行本节中的实例，你应该安装 Cassandra。可以在 GettingStarted 维基页面（http://wiki.apache.org/cassandra/GettingStarted）找到安装 Cassandra 的详细说明。

要继续这个实例，就在项目依赖关系中添加 `[clojurewerkz/cassaforte "1.1.0"]`，或用 `lein-try` 启动 REPL：

```
$ lein try clojurewerkz/cassaforte
```

为了连接 Cassandra 集群，创建并使用你的第一个键空间，就需要 `clojurewerkz.cassaforte.client`、`clojurewerkz.cassaforte.cql` 和 `clojurewerkz.cassaforte.query` 命名空间。

`clojurewerkz.cassaforte.client` 负责连接，另两个提供简单的接口来执行查询：

```
(require '[clojurewerkz.cassaforte.client :as client]
         '[clojurewerkz.cassaforte.cql :as cql]
         '[clojurewerkz.cassaforte.query :as q])

;; 连接集群中的 2 个节点
(client/connect! ["localhost" "another.node.local"])

;; 创建名为 `cassaforte_keyspace` 的键空间，使用
;; 简单复制策略，复制系数为 2
(cql/create-keyspace "cassaforte_keyspace"
                     (q/with {:replication
                              {:class "SimpleStrategy"
                              :replication_factor 2 }}))

;; 切换到该键空间
(cql/use-keyspace "cassaforte_keyspace")
```

现在可以创建表并插入数据，即调用 `clojurewerkz.cassaforte.cql` 命名空间的 `create-table` 和 `insert` 函数：

```
(cql/create-table "users"
                  (q/column-definitions {:name :varchar
                                         :city :varchar
                                         :age  :int
                                         :primary-key [:name]}))
```

接下来，在表中插入一些用户：

```
(cql/insert "users" {:name "Alex" :city "Munich" :age (int 26)})
(cql/insert "users" {:name "Robert" :city "Brussels" :age (int 30)})
```

可以用 `select` 查询来访问这些记录。例如，如果需要取出表中的所有用户，或在查询时使用 `limit`，可以执行：

```
;; 获取所有用户
(cql/select "users")
```

```
;; 获取前 10 个用户
(cql/select "users" (q/limit 10))
```

或者，如果需要按指定的 name 获取一个人的信息，可以添加 where 子句：

```
(cql/select "users" (q/where :name "Alex"))
```

# 讨论

Cassandra 是一个开源软件，实现了 Amazon 的里程碑式的 Dynamo 论文（http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf）中的许多思想。它是一个键值对数据库，不关心表和数据点之间的任何关系。Cassandra 是分布式的数据库，是为高可用性而设计的。为此，它在集群中复制数据。数据冗余存储在多个节点中。如果一个节点失效，数据仍然可以从不同的节点或多个节点中获取。

如果数据相当大，Cassandra 就更有意义了，因为它就是为分布式结构而设计的，可以扩大读写的规模，并很好地管理数据库的一致性和可用性。Cassandra 很好地处理了网络分断，所以即使有几个节点在一段时间内不可用，仍能读写数据，一直到网络分断恢复。如果数据相当小，在近期内不太会大量增长，而且需要对数据集执行许多特别的查询，那么 Cassandra 可能就不太适合。

一致性和可用性是可以调整的值。可以通过牺牲数据一致性，获得更好的可用性：在网络分断时，并非所有节点都保持了数据的最后快照，但仍然可以响应读和写的请求。相反，如果选择更强的一致性，延迟就会增加，因为读和写需要更多节点成功响应。如果对数据点没有冲突的写入，最终所有节点都将持有最新的值，最终一致性将得到保证。

像大多数据库一样，Cassandra 有独立数据库的概念（Cassandra 的术语是“键空间”）。每个键空间包含一些表（有时候称为“列族”）。表中包含行，行中包含列。每个列有键（列名）、值、写入时间戳和生存期。

Cassandra 使用两种不同的通信协议：较老的二进制协议名为 Thrift，以及 CQL（Cassandra 查询语言）。Cassaforte 中的所有操作背后都生成 CQL。下面两个例子说明这些操作如何在内部编译成 CQL：

```
(cql/select "users" (q/where :name "Alex"))
;; SELECT * FROM users WHERE name='Alex';

(cql/insert "users" {:name "Alex" :city "Munich" :age (int 26)})
;; INSERT INTO users (name, city) VALUES ('Munich', 26);
```

Cassandra 能做的远不止创建表和插入值。如果想更新数据库中的记录，可以调用 update 函数：

```
(cql/update "users"
             {:city "Berlin"}
             (q/where :name "Alex"))
```

从数据库中删除记录同样容易：

```
;; 将只删除一个用户
(cql/delete :users (q/where :name "Alex"))

;; 将删除名字与 IN 子句匹配的所有用户
(cql/delete :users (q/where :name [:in ["Alex" "Robert"]]))
```

如果想执行任意的 CQL 语句，不用 Cassaforte 的基于宏的 DSL，可以向 `client/execute` 函数传递一个字符串：

```
(client/execute
  "INSERT INTO users (name, city, age) VALUES ('Alex', 'Munich', 19);")
```

对于每次写入，可以指定一个可选的生存期，让数据在一段时间之后过期。这对于缓存和只想保留一段时间的数据（如用户会话）是有用的。例如，如果希望记录只生存 60 秒，可以执行：

```
(cql/insert "users" {:name "Alex" :city "Munich" :age (int 26)}
                    (q/using :ttl 60))
```

Cassandra 中另一个受欢迎的概念是分布式计数器。计数器列提供了一种有效的方式，对你需要的任何东西计数或求和。这是通过值的原子增加和减少操作来实现的。要从 Cassaforte 中创建一个带计数器的表，可以使用 `:counter` 列类型：

```
(cql/create-table :scores
                  (q/column-definitions {:username :varchar
                                         :score    :counter
                                         :primary-key [:username]}))
```

可以通过 `increment-by` 和 `decrement-by` 查询来增减计数器：

```
(cql/update :scores
            {:score (q/increment-by 50)}
            (q/where :name "Alex"))

(cql/update :scores
            {:score (q/decrement-by 5)}
            (q/where :name "Robert"))
```

## 参阅

- Cassaforte 的文档（http://clojurecassandra.info/）。

# 6.8　使用MongoDB

作者：Clinton Dreisbach

## 问题

希望使用保存在 MongoDB 中的数据。

## 解决方案

使用 Monger（http://clojuremongodb.info/）来连接 MongoDB，查找或操作数据。Monger 是 Java MongoDB 驱动的 Clojure 包装。

利用 Clojure 代码访问 Mongo 之前，必须运行要连接的 MongoDB 实例。如何在本地系统安装 MongoDB，请参考 MongoDB 的安装指南（http://docs.mongodb.org/manual/installation/）。

如果你已准备好编写 Clojure 的 MongoDB 客户端，就用 `lein-try` 启动 REPL：

```
$ lein try com.novemberain/monger
```

要连接 MongoDB，请使用 `monger.core/connect!` 函数。它将连接保存在动态 var `*mongodb-connection*` 中。如果希望取得连接并且不将它保存在动态 var 中，可以 `monger.core/connect` 函数，带上同样的选项：

```
(require '[monger.core :as mongo])

;; 连接 localhost
(mongo/connect! {:host "127.0.0.1" :port 27017})

;; 完成后断开连接
(mongo/disconnect!)
```

在连接成功后，可以方便地插入和查询文档：

```
(require '[monger.core :as mongo]
         '[monger.collection :as coll])
(import '[org.bson.types ObjectId])

;; 在 var *mongodb-database* 中设置数据库
(mongo/use-db! "mongo-time")

;; 插入一个文档
(coll/insert "users" {:name "Jeremiah Forthright" :state "TX"})

;; 插入一批文档
(coll/insert-batch "users" [{:name "Pete Killibrew" :state "KY"}
```

```
                                {:name "Wendy Perkins" :state "OK"}
                                {:name "Steel Whitaker" :state "OK"}
                                {:name "Sarah LaRue" :state "WY"}])

;; 查找所有文档，返回一个 com.mongodb.DBCursor
(coll/find "users")

;; 查找符合查询条件的所有文档，返回一个 DBCursor
(coll/find "users" {:state "OK"})

;; 查找文档，作为 Clojure 映射表返回
(coll/find-maps "users" {:state "OK"})
;; -> ({:_id #<ObjectId 520...>, :state "OK", :name "Wendy Perkins"}
;;     {:_id #<ObjectId 520...>, :state "OK", :name "Steel Whitaker"})

;; 查找一个文档，返回一个 com.mongodb.DBObject
(coll/find-one "users" {:name "Pete Killibrew"})

;; 查找一个文档，作为一个 Clojure 映射表返回
(coll/find-one-as-map "users" {:name "Sarah LaRue"})
;; -> {:_id #<ObjectId 520...>, :state "WY", :name "Sarah LaRue"}
```

# 讨论

MongoDB，特别是用了 Monger 后，可能是保存 Clojure 数据的自然选择。它以 BSON（二进制 JSON）格式保存数据，很符合 Clojure 自己的向量和映射表。有几种方法连接 Mongo，这取决于需要对连接进行多少定制，以及使用选项映射表还是 URI：

```
;; 默认连接 localhost 的 27017 端口
(mongo/connect!)

;; 连接另一个机器
(mongo/connect! {:host "192.168.1.100" :port 27017})

;; 利用更多复杂选项来连接
(let [options (mongo/mongo-options :auto-connect-retry true
                                   :connect-timeout 15
                                   :socket-timeout 15)
      server (mongo/server-address "192.168.1.100" 27017)]
  (mongo/connect! server options))

;; 通过 URI 来连接
(mongo/connect-via-uri! (System/genenv "MONGOHQ_URL"))
```

在插入数据时，为每个文档提供一个 _id 是可选的。如果在文档中没有，就会自动创建一个。但如果将来需要引用该文档，自己添加 _id 是有意义的：

```
(require '[monger.collection :as coll])
(import '[org.bson.types ObjectId])

(let [id (ObjectId.)
```

```
      user {:name "Lola Morales"}]
  (coll/insert "users" (assoc user :_id id))
  ;; 以后，通过 id 查找用户
  (coll/find-map-by-id "users" id))
;; -> {:_id #<ObjectId 521...>, :name "Lola Morales"}
```

在习惯用法中，Monger 设置为使用一个连接和一个数据库，因此 monger.core/connect! 和 monger.core/use-db! 设置了动态 var 来保存它们的信息。

但是，绕开它是很容易的。可以用 binding 在代码中明确设置它们。另外，可以用 monger.multi.collection 命名空间代替 monger.collection。monger.multi.collection 命名空间中的所有函数，都接受数据库作为第一个参数：

```
(require '[monger.core :as mongo]
         '[monger.multi.collection :as multi])

(mongo/connect!)

;; use-db! 接受一个字符串作为数据库，因为它是一个方便的函数，
;; 但对于 monger.multi.collection 和其他函数，我们需要用 get-db 来取得数据库
(let [stats-server (mongo/connect "stats.example.org")
      app-db (mongo/get-db "mongo-time")
      geo-db (mongo/get-db "geography")]

  ;; 在 stats 服务器中记录数据
  (binding [mongo/*mongodb-connection* stats-server]
    (multi/insert (mongo/get-db "stats") "access"
                  {:ip "127.0.0.1" :time (java.util.Date.)}))

  ;; 在应用 DB 中查找用户
  (multi/find-maps app-db "users" {:state "WY"})

  ;; 在地理 DB 中插入正方形
  (multi/insert geo-db "shapes"
                {:name "square" :sides 4
                 :parallel true :equal true}))
```

monger.collection 中的基本查找函数适用于简单的查询，但你很快会发现需要更复杂的查询，这时就需要 monger.query。这是针对 MongoDB 查询的领域特定语言：

```
(require '[monger.query :as q])

;; 查找用户，跳过前两个，取得接下来的三个
(q/with-collection "users"
  (q/find {})
  (q/skip 2)
  (q/limit 3))

;; 取得所有来自 Oklahoma 州的用户，按名字排序
;; 排序时必须用 array-map，可以保持键有序
(q/with-collection "users"
  (q/find {:state "OK"})
```

```
  (q/sort (array-map :name 1)))

;; 取得所有不是来自 Oklahoma 州或者名字以 "S" 开头的用户
(q/with-collection "users"
  (q/find {"$or" [{:state {"$ne" "OK"}}
                  {:name #"^S"}]}))
```

## 参阅

- Monger 的文档（http://clojuremongodb.info/）。
- CongoMongo（https://github.com/aboekhoff/congomongo），另一个使用MongoDB的Clojure库，你也许可以考虑。

# 6.9 使用Redis

作者：Jason Webb

## 问题

希望使用 Redis 中的数据。

## 解决方案

用 Carmine（https://github.com/ptaoussanis/carmine）连接 Redis，并与之交互。

> 要继续这个实例，应该先安装 Redis，并在本地运行。可以在官方 Redis 下载页面（http://redis.io/download）找到安装 Redis 的详细说明。如果系统是 Windows，可以看看 Microsoft Open Tech GitHub Redis 项目（https://github.com/MSOpenTech/redis）。

要继续这个实例，就在项目依赖关系中添加 `[com.taoensso/carmine "2.2.0"]`，或用 `lein-try` 启动 REPL：

```
$ lein try com.taoensso/carmine
```

要使用 Carmine，必须先定义连接规格说明：

```
(def server-connection {:pool {:max-active 8}
                        :spec {:host     "localhost"
                               :port     6379
                               ;;:password ""
                               :timeout  4000}})
```

Carmine 支持所有的 Redis 命令，名称（绝大部分）与 Redis 文档相符。用 `wcar` 函数和连

接规格说明 server-connection 来发送你熟悉和喜欢的 Redis 命令：

```
(require `[taoensso.carmine :as car :refer (wcar)])

(wcar server-connection (car/set "Nick" "Nack"))
;; -> "OK"
(wcar server-connection (car/get "Nick"))
;; -> "Nack"
(wcar server-connection (car/hset "founder" "name" "Tim"))
;; -> 0
(wcar server-connection (car/hset "founder" "age" 59))
;; -> 0
(wcar server-connection (car/hgetall "founder"))
;; -> [name Tim age 59]
```

传入多个命令会形成管道，结果作为向量一起返回：

```
(wcar server-connection (car/set "paddywhacks" 0)
                        (car/incr "paddywhacks")
                        (car/get "paddywhacks"))
;; -> ["OK" 1 "1"]
```

## 讨论

Redis 称自己是数据结构服务器。由于数据结构与 Clojure 中的核心数据结构类似，它们很适合一起解决各种问题。Redis 的速度和键 / 值存储，让它特别适合实现缓存和应用内存化（稍后进一步讨论）。

通过将 wcar 的调用包装在宏中，传入连接规格说明，就可以消除一些引用：

```
(defmacro wcar* [& body] `(car/wcar server-connection ~@body))

(wcar* (car/set "Nick" "Nack"))
;; -> "OK"
(wcar* (car/get "Nick"))
;; -> "Nack"
```

序列化是自动处理的，在大多数情况下有效。只要传入你想保存的数据，Carmine 将自动完成序列化 / 反序列化：

```
(wcar* (car/set "some-key" {:event "An Event", :timestamp (new java.util.Date)})
       (car/get "some-key"))
;; -> [OK {:event An Event, :timestamp #inst "2013-08-18T21:31:33.993-00:00"}]
```

只要你坚持使用 Clojure 核心数据类型，大多数情况下都会得到理想的结果。但是，如果需要支持保存定制的数据类型，就要利用底层的序列化库，名为 Nippy。更多信息请参见 Nippy 的 GitHub 项目（https://github.com/ptaoussanis/nippy）。

Redis 很适合作为内存化存储后端。显然，在评估内存解决方案时，有一些严肃的折中要考

虑，如 core.cache 库。但是如果方案正确，将获得很大的提升。例如，请考虑将一个函数内存化，它访问一个外部的 Web 服务，取得当前的天气信息。通过很少的工作，多个服务器就能共享最新的数据，甚至让过时的数据自动过期并刷新。下面的例子就是这样一种情况：

```
(defn redis-memoize
  "Convert a function to one that is memoized using Redis as storage."
  [key-prefix ttl-seconds connection-spec f]
  (fn [& args]
    (let [key-name [key-prefix args]]
      (if-let [found-result (wcar connection-spec (car/get key-name))]
        found-result
        (let [new-result (apply f args)]
          (wcar connection-spec (car/set key-name new-result)
                                (car/expire key-name ttl-seconds))
          new-result)))))
```

值得一提的是，这里作了一些假定。首先，它假定被内存化的函数所带的参数是 Nippy 支持的（请看前面序列化的例子）。其次，它假定内存化的数据应该在指定的秒数后过期。要使用 redis-memoize，只要传入一个函数。下面设计极为精巧的例子使用了前面定义的 server-connection：

```
(defn square [x]
  (printf "Ran square for: %s\n" x)
  (* x x))

(def redis-squared
  (redis-memoize "squared" 10 server-connection square))

(redis-squared 99)
;; -> Ran square for: 99
;; -> 9801
(redis-squared 99)
;; -> 9801
```

除了前面列出的特征，Carmine 还包括（但不限于）消息队列，分布式锁，Ring 会话库，甚至 DynamoDB 的实现（在本书编写时还处于 alpha 阶段）。这些特征超出了本实例的范围，但它们有很好的文档，很容易使用。更多信息请参考 Carmine 的 GitHub 项目（https://github.com/ptaoussanis/carmine）。

## 参阅

- Carmine 的 GitHub 项目（https://github.com/ptaoussanis/carmine），了解关于 Carmine 的更多信息。
- 官方 Redis 文档（http://redis.io/commands），了解完全的 Redis 命令清单。
- Nippy 的 GitHub 项目（https://github.com/ptaoussanis/nippy），了解序列化的信息。
- Clojure 的核心文档（http://clojuredocs.org/clojure_core/clojure.core/memoize/clojure.core/memoize），了解 memoize 函数的文档。

# 6.10 连接Datomic数据库

作者：Robert Stuttaford

## 问题

需要连接 Datomic 数据库。

## 解决方案

开始之前，在项目依赖关系中添加 `[com.datomic/datomic-free "0.8.4218"]`，或用 `lein-try` 开始 REPL：

```
$ lein try com.datomic/datomic-free
```

要创建并连接到一个内存数据库，就用 `database.api/create-database` 和 `datomic.api/connect`：

```
(require '[datomic.api :as d])

(def uri "datomic:mem://sample-database")

(d/create-database uri)
;; -> true

(def conn (d/connect uri))

conn
;; -> #<LocalConnection datomic.peer.LocalConnection@49384d99>
```

建立连接后，就可以利用它，通过 `datomic.api/db` 取得数据库的值。这个值将用于查询数据库：

```
(def db (d/db (d/connect uri)))

db
;; -> datomic.db.Db@7b7fea26
```

也可以通过 `datomic.api/transact` 利用连接来处理数据事务：

```
;; 让 schema 支持事务，目的是接下来的大事情
(def my-great-schema []) ; 这个向量是有意为空的
(d/transact (d/connect uri) my-great-schema)
```

## 讨论

在解决方案中你会注意到，我们不仅连接了数据库，而且创建了它。这在使用内存数据库

时是很常见的，因为在新的 JVM 中没有内存数据库。如果数据库已存在，就不一定要调用 `create-database`，但这样做比较安全，因为 `create-database` 是幂等的，如果数据库已存在，它会返回 `false`。如果连接的数据库不在内存中，就需要相关的事务管理器和存储服务进入运行状态。

`d/connect` 的返回值会在查询数据库中的值或执行数据事务时用到。在读取事务日志，消费事务报告队列，执行管理任务，如要求建立索引、垃圾收集、释放连接相关的资源时，也会用到它。连接是线程安全的，由 URI 在内部缓存，所以不需要自己建立连接池。对同一个 URI 建立许多连接也没有性能开销。

### 存储服务

Datomic 事务管理器进程对于并发连接的进程数有限制。Datomic Free 对每个事务管理器限制两个连接。对于非分布式应用，这可能已足够。如果你要构建大型服务，那就需要 Datomic Pro 的许可证，支持更多连接。

有一些选项是针对后端为 Datomic 的存储服务的。三个选项是内建的，其他选项用到了外部服务。Datomic Free 包括对内存和 `:free` 存储后端的访问。Datomic Pro 和 Pro Starter Edition 包括对所有服务的访问。

#### 内建的存储选项

以下是内建的存储选项。

- 本地内存：`"datomic:mem://[db-name]"`；
- Free，使用 Datomic Free，有两个连接的限制：`"datomic:free://host[:port]/[db-name]"`；
- Dev，使用 Datomic Pro，受许可的连接数限制：`"datomic:dev://host[:port]/[db-name]"`。

Free 版和 Dev 版也可以配置使用别的端口作为存储：`"datomic:free://host[:port]/[db-name]?h2-port=[port]&h2-web-port=[port]"`。默认情况下，这些端口依次排在事务管理器端口之后。

#### 外部存储服务选项

还有几个外部存储选项。

- DynamoDB：`"datomic:ddb://[aws-region]/[dynamodb-table]/[db-name]? aws_access_key_id=[XXX]&aws_secret_key=[YYY]"`；
- Riak：`"datomic:riak://host[:port]/bucket/dbname[?interface=http|proto buf]"`（默认是 protobuf）；
- Couchbase：`"datomic:couchbase://host/bucket/dbname[?password=xxx]"`；
- Infinispan：`"datomic:inf://[cluster-member-host:port]/[db-name]"`；
- SQL：`"datomic:sql://[db-name][?jdbc-url]"`。

对于 SQL 存储服务，可以用映射表格式代替字符串格式。如果要指定不能嵌入 URI 字符串中的对象，如 DataSource，这是有用的。SQL 映射表的格式是：

```
{:protocol :sql                   ;; 关键字或字符串
 :db-name "[db-name]"             ;; 关键字或字符串
 :data-source aDataSourceObject
  ;; OR
 :factory aCallableReturningConnection}
```

## 参阅

- 6.11 节“为 Datomic 数据库定义数据模式”。
- 6.12 节“向 Datomic 写入数据”。
- Datomic Pro Starter Edition（http://blog.datomic.com/2013/11/datomic-pro-starter-edition.html），了解免费访问所有存储和 Datomic 控制台。

# 6.11　为Datomic数据库定义数据模式

作者：Robert Stuttaford

## 问题

需要定义数据在 Datomic 中的模型。例如，需要对用户和用户组建模，以某种方式关联起来。

## 解决方案

Datomic 数据模式是以*属性*的方式来定义的。了解它最容易的方法是直接看例子。

要继续本实例，请完成 6.10 节“连接 Datomic 数据库”中的步骤。然后，你就有了一个内存数据库和连接 conn。

考虑用户可能有的属性：

- email 地址，在数据库中必须唯一；
- 姓名，对它索引以便快速查找；
- 任意多个角色（访客、作者和编辑）。

要定义这个数据模式，就要创建一个向量，其中包含 email、姓名和角色的属性映射表，并插入三种不变的角色：

```
(def user-schema
  [{:db/doc "User email address"
```

```
  :db/ident :user/email
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one
  :db/unique :db.unique/identity
  :db/id #db/id[:db.part/db]
  :db.install/_attribute :db.part/db}

 {:db/doc "User name"
  :db/ident :user/name
  :db/valueType :db.type/string
  :db/cardinality :db.cardinality/one
  :db/index true
  :db/id #db/id[:db.part/db]
  :db.install/_attribute :db.part/db}

 {:db/doc "User roles"
  :db/ident :user/roles
  :db/valueType :db.type/ref
  :db/cardinality :db.cardinality/many
  :db/id #db/id[:db.part/db]
  :db.install/_attribute :db.part/db}

 [:db/add #db/id[:db.part/user] :db/ident :user.roles/guest]
 [:db/add #db/id[:db.part/user] :db/ident :user.roles/author]
 [:db/add #db/id[:db.part/user] :db/ident :user.roles/editor]])
```

我们定义用户组的属性包括：

- UUID，在数据库中必须唯一；
- 名称，对它索引以便快速查找；
- 任意多个相关的用户。

这样定义用户组：

```
(def group-schema
  [{:db/doc "Group UUID"
    :db/ident :group/uuid
    :db/valueType :db.type/uuid
    :db/cardinality :db.cardinality/one
    :db/unique :db.unique/value
    :db/id #db/id[:db.part/db]
    :db.install/_attribute :db.part/db}

   {:db/doc "Group name"
    :db/ident :group/name
    :db/valueType :db.type/string
    :db/cardinality :db.cardinality/one
    :db/index true
    :db/id #db/id[:db.part/db]
    :db.install/_attribute :db.part/db}

   {:db/doc "Group users"
    :db/ident :group/users
```

```
        :db/valueType :db.type/ref
        :db/cardinality :db.cardinality/many
        :db/id #db/id[:db.part/db]
        :db.install/_attribute :db.part/db}])
```

最后，用 `transact` 函数将两个数据模式定义通过连接传入数据库：

```
(require '[datomic.api :as d])

@(d/transact (d/connect "datomic:mem://sample-database")
             (concat user-schema group-schema))
;; -> {:db-before datomic.db.Db@25b48c7b,
;;     :db-after datomic.db.Db@5d81650c,
;;     :tx-data [#Datum{:e ... :a ... :v ... :tx :added true}, ...],
;;     :tempids {-... ..., ...}}
```

## 讨论

Datomic 数据模式表示为 Clojure 数据，并在一个事务中添加到数据库，就像存储的其他数据一样。`:db.install/_attribute :db.part/db` 键 / 值对被事务管理器使用，让系统的其他部分能使用这个数据模式。

数据模式放在 `:db.part/db` 数据库分区中，该分区专为数据模式保留。所有用户数据放在用户分区中，要么是默认的 `:db.part/user`，要么是定制的分区。分区有助于优化索引如何排序数据，从而有助于优化查询。数据模式实体要求至少提供 `:db/ident`、`:db/valueType` 和 `:db/cardinality` 值。

除了数据模式，Datomic 对任何实体都不强制属性必须组合使用。Datomic 只要求必须先定义数据模式，在运行时强制满足类型和唯一性约束。

在数据模式 `:db/ident` 的值中使用命名空间，有助于对实体分类（如 `:user/email` 中的 `user`）。Datomic 对命名空间不作任何特殊处理，使用它们是可选的。`:db/valueType` 有几个选项，在表 6-1 中列出。

**表6-1：db/值类型选项**

| | | |
|---|---|---|
| :db.type/keyword | :db.type/string | :db.type/long |
| :db.type/boolean | :db.type/bigint | :db.type/float |
| :db.type/double | :db.type/bigdec | :db.type/instant |
| :db.type/ref | :db.type/uuid | :db.type/uri |
| :db.type/bytes | | |

其语义的详尽列表，参见 Datatomic 数据模式文档（http://docs.datomic.com/schema.html）。

带有 `:db/valueType :db.type/ref` 的属性只能用其他实体作为它们的值。可以用这个类型来建模实体间的关系。Datomic 不强制哪些实体与给定的 `:db/valueType :db.type/ref` 属性关

联。任何其他实体都可以关联，这意味着实体可以与自己关联！

可以用 `:db/valueType :db.type/ref` 和一些单独的 `:db/ident` 值来建模枚举值，例如前面定义的用户角色。这些枚举值其实不是数据模式，它们是普通的实体，带有唯一属性 `:db/ident`。实体的 `:db/ident` 值可以作为该实体的简写，在事务和查询中，可以用这个值代替实体的 `:db/id` 值。

带有 `:db/valueType :db.type/ref` 和 `:db/unique` 值的属性是隐式索引的，就像定义中添加了 `:db/index true` 一样。

也可以对字符串属性使用 Lucene 的全文本索引，使用 `:db/fulltext true` 和系统在 Datalog 中定义的 `fulltext` 函数。

在 `:db/unique` 中指定唯一性约束时有两个选项。

`:db.unique/value`
: 对不同的实体 ID，不允许尝试插入重复的值。

`:db.unique/identity`
: 指定属性值对每个实体是唯一的，并支持*更新插入*。对临时实体 ID 插入重复的值的任何尝试，都将导致所有与该临时 ID 关联的属性与数据库中原来的实体合并。

如果要建模的实体带有子实体，只存在于那些实体的上下文中，例如订单中的订单项或产品的变种，就可以用 `:db/isComponent` 来简化这种子实体的处理。它只能用于 `:db.type/ref` 类型的属性。

如果在事务中使用 `:db.fn/retractEntity` 函数，被回撤实体中这种属性对应的值实体也会被回撤。而且，如果用 `d/touch` 来实现实体映射表中的所有惰性键，组件实例也会被实现。回撤和实现行为都是递归的。

默认情况下，Datomic 保存属性过去所有的值。如果对某个属性不希望保存过去的值，就用 `:db/noHistory true`，让 Datomic 丢弃以前的值。使用这个属性很像使用传统的就地更新（update-in-place）数据库。

## 参阅

- 6.12 节“向 Datomic 写入数据”，了解关于 datoms（schemas!）事务的更多信息。

# 6.12　向Datomic写入数据

作者：Robert Stuttaford

## 问题

需要向 Datomic 数据库添加数据。

## 解决方案

用 Datomic 连接来实现数据事务。

要继续本实例，请完成 6.10 节“连接 Datomic 数据库”和 6.11 节“为 Datomic 数据库定义数据模式”中的步骤。

然后，你就有了连接 `conn` 和数据模式，可以插入数据：

```
(require '[datomic.api :as d :refer [q db]])

(def tx-data [{:db/id (d/tempid :db.part/user)
               :user/email "fowler@acm.org"
               :user/name "Martin Fowler"
               :user/roles [:user.roles/author :user.roles/editor]}])
@(d/transact conn tx-data)

(q '[:find ?name
     :where [?e :user/name ?name]]
   (:db-after tx-result))
;; -> #{["Martin Fowler"]}
```

## 讨论

这种基于映射表的数据表示语法，将扩展成一系列 `:db/add` 语句。下面的事务与前面的等价：

```
(def new-id (d/tempid :db.part/user))
new-id
;; -> #db/id[:db.part/user -1000013]

(def tx-data2 [[:db/add new-id :user/email "ryan@cognitect.com"]
               [:db/add new-id :user/name "Ryan Neufeld"]
               [:db/add new-id :user/roles [:user.roles/author
                                            :user.roles/editor]]])

(def tx-result @(d/transact conn tx-data2)) ;; Keep this for later...

(q '[:find ?name
     :where [?e :user/name ?name]]
   (db conn))
;; -> #{["Ryan Neufeld"] ["Martin Fowler"]}
```

当然，你可以使用这样的语句，也可以用解决方案中的映射表语法，还可以混合使用。这是对多个条目执行事务的方法（例如，`(d/transact conn [person1-map person2-map])`）。

你会注意到，映射表和扩展后的形式之间有一点差别，即缺少针对 :db/id 键的 :db/add 语句。在扩展后的形式中，这个值紧跟在动作（:db/add）之后，对同一个实体的所有相关属性，这个值必须相等。如果将实体指定为映射表，只要提供一个 ID，事务处理器会透明地加在每个属性前面。

什么 ID 合适？新的实体都会指定负的临时 ID 值，可以在事务中用它对关系建模。在事务成功时，所有临时 ID 都会被赋予数据库中正的 ID 值。在代码中，正确的方法是用 datomic.api/tempid 函数得到临时 ID。datomic.api/tempid 函数接受一个分区关键字和一个可选的 ID 作为参数，对于大多数情况，:db.part/user 就足够了。

在处理不可执行的数据时，需要用数据字面量的形式来表示临时 ID。字面量 #db/id [:db.part/user] 等价于 (d/tempid :db.part/user)。如果将事务数据保存在 .edn 文件中，这种形式就特别有用。在数据模式定义时常常遇到这种情况。同样，在代码中应该使用 d/tempid——因为 #db/id 字面量会在编译时求值一次，这意味着如果预期 ID 值在两次运行时会不同，代码就会失效，因为它永远只有一个值。

请考虑我们的示例文件，user-bootstrap.edn：

```
[{:db/id #db/id [:db.part/user]
  :user/email "fowler@acm.org"
  :user/name "Martin Fowler"
  :user/roles [:user.roles/author :user.roles/editor]}]
```

在事务完成时，你会收到一个完成的 future 对象。如果你喜欢异步的事务，可以用 d/transact-async 代替，它将立即返回 future 对象。在这种情况下，就像所有 future 对象一样，如果你对它去引用（dereference），它将阻塞直至事务完成。不管哪种方式，对 future 去引用将返回一个映射表，包含以下 4 个键。

:db-before
: 在事务执行之前数据库的值。

:db-after
: 在事务执行之后数据库的值。

:tx-data
: 所有执行事务的 datom 构成的向量。

:tempids
: 临时 ID 到数据库中 ID 的映射，事务中的每个临时 ID 有一个表项。

在事务之后，可以用 :db-after 数据库来直接查询该数据库：

```
(def db-after-tx (:db-after tx-result))

(q '[:find ?name :in $ ?email :where
     [?entity :user/email ?email]
     [?entity :user/name ?name]]
   db-after-tx
   "fowler@acm.org")
;; -> #{["Martin Fowler"]}
```

对于你关心的新实体，可以通过 `:tempids` 映射表找到数据库中的 ID，就像在 SQL 数据库中取得最后插入的 ID 一样。调用 `datomic.api/resolve-tempid` 函数，参数是 `:db-after` 的值、`:tempids` 的值和最初的临时 ID，可以取得实现后的 ID：

```
(d/resolve-tempid db-after-tx (:tempids tx-result) new-id)
;; -> 17592186045421
```

## 参阅

- 6.11 节“为 Datomic 数据库定义数据模式”。
- 6.13 节“从 Datomic 数据库中删除数据”。
- 6.14 节“尝试 Datomic 事务而不提交”。

# 6.13 从Datomic数据库中删除数据

作者：Robert Stuttaford

## 问题

需要从 Datomic 数据库中删除数据。

## 解决方案

要删除一个属性的值，应该在事务中使用 `:db/retract` 操作。

要继续本实例，请完成 6.10 节“连接 Datomic 数据库”和 6.11 节“为 Datomic 数据库定义数据模式”中的步骤。然后，你就有了连接 `conn` 和数据模式，可以插入数据。

开始先添加一个用户 Barney Rubble，并验证他有 email 地址：

```
(def new-id (d/tempid :db.part/user))

(def tx-result @(d/transact conn
                            [{:db/id new-id
                              :user/name "Barney Rubble"
                              :user/email "barney@example.com"}]))
```

```
(def after-tx-db (:db-after tx-result))

(def barney-id (d/resolve-tempid after-tx-db
                                 (:tempids tx-result)
                                 new-id))

barney-id
;; -> 17592186045429

(d/q '[:find ?email :in $ ?entity-id :where
       [?entity-id :user/email ?email]]
     after-tx-db
     barney-id)
;; -> #{["barney@rubble.me"]}
```

要撤销 Barney 的 email，就在事务中执行 `:db/retract` 操作：

```
(def retract-tx-result @(d/transact conn [[:db/retract barney-id
                                           :user/email "barney@example.com"]]))

(def after-retract-db (:db-after retract-tx-result))

(d/q '[:find ?email :in $ ?entity-id :where
       [?entity-id :user/email ?email]]
     after-retract-db
     barney-id)
;; -> #{}
```

要撤销整个实体，就使用内建的事务管理器函数 `:db.fn/retractEntity`：

```
(def retract-entity-tx-result
  @(d/transact conn [[:db.fn/retractEntity barney-id]]))

(def after-retract-entity-db (:db-after retract-entity-tx-result))

(d/q '[:find ?entity-id :in $ ?name :where
       [?entity-id :user/name ?name]]
     after-retract-entity-db
     "Barney Rubble")
;; -> #{}
```

## 讨论

在使用 `:db/retract` 时，你提供要撤销的值，以便在多基数属性的情况下，明确属性的值集合中要撤销哪个值。不论基数如何，如果提供的值不在存储器中，什么都不会撤销。这意味着必须知道要撤销的值是什么，不能只提供实体的 ID 和属性，就撤销该属性的所有值。

如果被撤销值的属性没有使用 `:db/noHistory`，你就能够查询过去数据库的值，找到该属性过去的值。

如果被撤销值的属性使用了 `:db/noHistory`，数据就被永久删除了。

在使用 `:db.fn/retractEntity` 时，该实体的所有属性的所有值都会被撤销，所有 `:db/ref` 属性以实体作为值，也会被撤销。撤销实体的所有组件实体，将会被递归撤销。

你会发现真正的实体 ID 本身没有被撤销，但没有任何属性与它关联。这是因为实体一旦创建，它就不能撤销。但是，删除所有属性以及对该实体的引用，效果等同于永久地删除它！

如果出于法律考虑，或者面对的数据超出了领域特定的保存期，因此需要永久地删除数据，请使用 excision（http://blog.datomic.com/2013/05/excision.html）永久地删除数据。

## 参阅

- Datomic 博客文章（http://blog.datomic.com/2013/05/excision.html）介绍了 excision 的功能。

# 6.14 尝试Datomic事务而不提交

作者：Robert Stuttaford

## 问题

希望在使用 Datalog 或实体 API 提交事务之前，先测试事务。

## 解决方案

像平常一样构建事务，但不是调用 `d/transact` 或 `d/transact-async`，而是用 `d/with` 得到一个内存数据库，其中包含事务所产生的变更。

要继续本实例，请完成 6.10 节“连接 Datomic 数据库”和 6.11 节“为 Datomic 数据库定义数据模式”中的步骤。然后，你就有了连接 `conn` 和数据模式，可以插入数据。

首先，向数据库中添加关于 Fred Flintstone 的数据。由于是在公元前 4000 年左右，Fred 没有 email，但至少我们知道他的姓名：

```
(require '[datomic.api :as d])

(def new-id (d/tempid :db.part/user))


(def tx-result @(d/transact conn
                            [{:db/id new-id
                              :user/name "Fred Flintstone"}]))
```

快速跳到今天：在冰中冻了 6000 年后，Fred 被解冻了，他有了自己第一个 email 地址。准备一个事务，为 Fred 实体添加 email：

```
;; 从最初的事务中取得 Fred 的 ID
(def fred-id (d/resolve-tempid (:db-after tx-result)
                               (:tempids tx-result)
                               new-id))

fred-id
;; -> 17592186045421

(def add-freds-email-tx [[:db/add fred-id
                          :user/email "twinkletoes@example.com"]])
```

现在，准备一个内存数据库，执行新事务。首先，取得当前数据库的值作为基础，然后创建内存数据库。最后，取得 `:db-after` 值，这样就可以测试 email 值是否添加正确：

```
(defn db-with
  "Return a new database with tx applied"
  [db tx]
  (-> (d/with db tx)
      :db-after))

(def db-after (db-with (d/db conn) add-freds-email-tx))
```

比较当前数据库中 Fred 的 email 和内存数据库中 Fred 的 email：

```
(defn users-email
  "Retrieve a user's email given the user's name."
  [db name]
  (-> (d/q '[:find ?email
             :in $ ?name
             :where
             [?entity :user/name ?name]
             [?entity :user/email ?email]]
           db
           name)
        ffirst))

(users-email db-after "Fred Flintstone")
;; -> "twinkletoes@example.com"

(users-email (d/db conn) "Fred Flintstone")
;; -> nil
```

可以看到，当前数据库没有受到这个事务的影响，但在 `db-after` 中的数据库现在显示出新值。

## 讨论

`d/with` 产生的数据库可以用于所有其他接受数据库的 API 函数，包括 `d/with` 本身。这意

味着你可以执行一次又一次的事务，而不必将它们提交给事务管理器！

Datomic 如此强大，有一点是因为它能将数据库作为一个值。出于这个原因，我们写的辅助函数接受数据库作为参数，而不是连接。现在，不仅可以查询当前的数据库，而且也可以查询该数据库的其他值。

## 参阅

- 6.12 节“向 Datomic 写入数据”，了解关于数据事务的更多一般信息。

# 6.15　遍历Datomic索引

作者：Alan Busby 和 Ryan Neufeld

## 问题

希望快速执行简单的 Datomic 查询。

## 解决方案

用 `datomic.api/datoms` 函数直接访问数据库中的核心 Datomic 索引。

要继续本实例，请完成 6.10 节“连接 Datomic 数据库”和 6.11 节“为 Datomic 数据库定义数据模式”中的步骤。然后，你就有了连接 `conn` 和数据模式，可以插入数据。

例如，要很快找到有指定属性和值集的实体，就调用 `datomic.api/datoms`，指定 `:avet` 索引（属性、值、实体、事务）以及期望的属性和值：

```
(require '[datomic.api :as d])

(d/transact conn [{:db/id (d/tempid :db.part/user)
                   :user/name "Barney Rubble"
                   :user/email "barney@example.com"}])

(defn entities-with-attr-val
  "Return entities with a given attribute and value."
  [db attr val]
  (->> (d/datoms db :avet attr val)
       (map :e)
       (map (partial d/entity db))))


(def barney (first (entities-with-attr-val (d/db conn)
                                           :user/email
                                           "barney@example.com")))
```

```
(:user/email barney)
;; -> "barney@example.com"
```

这只适用于 :db/index 为 true 或 :db/unique 不为 nil 的属性。

要快速取得实体的所有属性，就用 :eavt 排序的索引：

```
(defn entities-attrs
  "Return attrs of an entity"
  [db entity]
  (->> (d/datoms db :eavt (:db/id entity))
       (map :a)
       (map (partial d/entity db))
       (map :db/ident)))

(entities-attrs (d/db conn) barney)
;; -> (:user/email :user/name)
```

要快速找到通过 :db.type/ref 来引用指定实体的所有实体，就用 :vaet 排序的索引：

```
;; 添加一个人，引用 :user.roles/author 角色
(d/transact conn [{:db/id (d/tempid :db.part/user)
                   :user/name "Ryan Neufeld"
                   :user/email "ryan@rkn.io"
                   :user/roles [:user.roles/author :user.roles/editor]}])

(defn referring-to
  "Find all entities referring to an entity as a certain attribute."
  [db entity]
  (->> (d/datoms db :vaet (:db/id entity) )
       (map :e)
       (map (partial d/entity db))))

(def author-entity (d/entity (d/db conn) :user.roles/author))

;; 具有 :user.roles/author 角色的所有用户的姓名

(map :user/name (referring-to (d/db conn) author-entity))
;; -> ("Ryan Neufeld")
```

## 讨论

对于简单的查找查询，如“按属性查找”或“按值查找”，在性能上没有超过 Datomic 的原始索引的。datomic.api/datoms 接口提供对所有 Datomic 索引的访问，让你能深入任何层级，找出正好是你需要的数据。

像大多数 Datomic 函数一样，`datoms` 接受 `db` 作为第一个参数。在我们的例子中，以及本书的其他地方，你都会注意到，我们也接受数据库作为一个值，而不是一个连接，因为这种习惯允许 API 用户对同一个数据库值执行各种操作。你应该自己试一下。

`datoms` 的第二个参数表明要访问的具体索引。每个值都是字母 `e`（实体）、`a`（属性）、`v`（值）和 `t`（事务）的不同排列。索引中字母的次序表明它如何被索引。例如，`:eavt` 应该按实体遍历，然后是属性，以此类推。四种索引以及它们包含的内容如下：

`:eavt`
: 包含所有 datom 的、实体为先的索引。这个索引提供的数据库视图很像传统的关系数据库。

`:aevt`
: 包含所有 datom 的、先属性后实体的索引。这个索引提供数据库的列访问，很像数据仓库。

`:avet`
: 属性 – 值索引，只包含 `:db/index` 为 `true` 的属性。对于查找索引非常有用（例如，“我需要 email 为 foo@example.com 的实体”）。

`:vaet`
: 值优先的索引，只包含 `:db.type/ref` 的值。这是非常有趣的索引，可以让数据有点像图数据库。

在指定了索引次序后，就可以选择提供任意个组件来预遍历该索引。这样做是为了减少返回元素的数目。例如，对 AVET 遍历只指定属性组件，将返回具有那种属性的所有实体。指定属性和值组件，将只返回具有指定属性 / 值对的实体。

`datoms` 返回的是一个 `Datum` 对象流。每个 `Datum` 对象都作为函数响应 `:a`、`:e`、`:t`、`:v` 和 `:added` 等关键字。

## 参阅

- 6.12 节“向 Datomic 写入数据”。

第 7 章

# Web应用

## 7.0 简介

如今，Web 应用开发是许多编程语言的谋生手段，Clojure 也不例外。在 2013 年度 Clojure 现状调查（http://cemerick.com/2013/11/18/results-of-the-2013-state-of-clojure-clojurescript-survey/）中，对于"你在什么领域使用 Clojure 或 ClojureScript？"这样的问题，Web 开发是排名第一的答案。

现在许多 Clojure Web 开发社区都以 Ring（https://github.com/ring-clojure/ring）为中心，它是一个 HTTP 服务器库，很像 Ruby 的 Rack（http://rack.github.io/）。在 7.1 节"Ring 简介"之后，我们提供了各种补充实例，让你能很快实现高速开发。

在 Ring 之后，本章介绍了目前可用的其他 Clojure Web 开发生态系统，包括一些模板和 HTML 操作库，以及另一些 Web 框架。

## 7.1 Ring简介

作者：Adam Bard

### 问题

需要用 Clojure 编写一个 HTTP 服务。

## 解决方案

Clojure 没有内建的 HTTP 服务器，但服务基本同步 HTTP 请求的事实标准是 Ring 库。

要继续这个实例，请复制 https://github.com/clojure-cookbook/ringtest 代码库，并覆写 src/ringtest.clj：

```
(ns ringtest
  (:require
    [ring.adapter.jetty :as jetty]
    clojure.pprint))

;; 返回收到的请求（经过美化打印）
(defn handler [request]
  {:status 200
   :headers {"content-type" "text/clojure"}
   :body (with-out-str (clojure.pprint/pprint request))})

(defn -main []
  ;; 在端口 3000 运行服务器
  (jetty/run-jetty handler {:port 3000}))
```

## 讨论

Ring（https://github.com/ring-clojure/ring）是许多 Clojure Web 应用的基础。它提供了底层的、简单的请求 / 响应 API，其中请求和响应是普通的 Clojure 映射表。Ring 应用是围绕“处理函数”建立的：这些函数接受请求并返回响应。前面的例子定义了一个简单的处理函数，只是把收到的请求返回给响应。

基本的响应映射表包含三个键：`:status` 是返回的状态码，`:headers` 是可选的“字符串 – 字符串”映射表，包含需要的响应头，`:body` 是字符串，包含需要的响应体。这里，`:status` 是 `200`，`:body` 是美化打印过的请求字符串。因此，在作者的机器上访问 URL http://localhost:3000/test/path/?qs=1 得到了下面的响应，展示了请求的结构：

```
{:ssl-client-cert nil,
 :remote-addr "0:0:0:0:0:0:0:1",
 :scheme :http,
 :request-method :get,
 :query-string "qs=1",
 :content-type nil,
 :uri "/test/path/",
 :server-name "localhost",
 :headers
 {"accept-encoding" "gzip,deflate,sdch",
  "connection" "keep-alive",
  "user-agent"
  "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_4) AppleWebKit/537.36
  (KHTML, like Gecko) Chrome/28.0.1500.71 Safari/537.36",
  "accept-language" "en-US,en;q=0.8",
```

```
 "accept"
 "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
 "host" "localhost:3000",
 "cookie" ""},
:content-length nil,
:server-port 3000,
:character-encoding nil,
:body #<HttpInput org.eclipse.jetty.server.HttpInput@43efe432>}
```

可以看到这很全面，但是很底层，请求中突出的特点解析成 Clojure 数据结构，没有额外的抽象。通常会利用其他的代码或库，从这个数据结构中提取有意义的信息。

`Jetty` 用于运行嵌入式的 Jetty 服务器。Ring 也带有适配器，可以作为 servlet 运行在所有 Java servlet 容器中。

请注意，调用 `run-jetty` 是同步的，只要服务器在运行，它就不会返回。如果从 REPL 中调用，应该用一个 future 对象包装它（或采用其他并发机制），这样服务器就会运行在其他线程中，REPL 就不会失去响应。

## 参阅

- Ring 的 GitHub 代码库（https://github.com/ring-clojure/ring）。

# 7.2 使用Ring中间件

作者：Adam Bard

## 问题

希望构建一个转换器，自动应用于 Ring 的请求或响应。例如，Ring 提供一些请求字符串，但你更希望处理解析过的映射表。

## 解决方案

因为 Ring 处理普通的 Clojure 数据和函数，可以简单地将中间件定义为返回函数的函数。在这里，定义一个中间件来修改请求，在请求中添加解析过的请求字符串，再将它传递给处理函数。

要继续这个实例，请复制 https://github.com/clojure-cookbook/ringtest 代码库，并覆写 src/ringtest.clj：

```
(ns ringtest
  (:require
    [ring.adapter.jetty :as jetty]
    [clojure.string :as str]
```

```
    clojure.pprint))

(defn parse-query-string
  "Parse a query string to a hash-map"
  [qs]
  (if (> (count qs) 0) ; 不要操作 nil 或空字符串
    (apply hash-map (str/split qs #"[&=]"))))

(defn wrap-query
  "Add a :query parameter to incoming requests that contains a parsed
  version of the query string as a hash-map"
  [handler]
  (fn [req]
    (let [parsed-qs (parse-query-string (:query-string req))
          new-req (assoc req :query parsed-qs)]
      (handler new-req))))

(defn handler [req]
  (let [name (get (:query req) "name")]
    {:status 200
     :body (str "Hello, " (or name "World"))}))

(defn -main []
  ;; 在端口 3000 运行服务器
  (jetty/run-jetty (wrap-query handler) {:port 3000}))
```

## 讨论

因为 Ring 处理函数操作普通的 Clojure 映射表，所以很容易写一个中间件来包装处理函数。这里，我们写了一个中间件来修改请求，再将它传递给处理函数。如果原始请求像这样：

```
{:query-string "x=1&y=2"
 ; ... and the rest
 }
```

那么处理函数收到的请求就变成：

```
{:query-string "x=1&y=2"
 :query {"x" "1" "y" "2"}
 ; ... and the rest
 }
```

不过，你不需要为此编写自己的中间件。Ring 提供了一些中间件让你加到应用中，其中包括名为 `wrap-params` 的中间件，它做的事和我们刚才写的中间件一样，但更好。根据 Ring 的文档（http://clojuredocs.org/ring/ring.middleware.params/wrap-params）：

> wrap-params
>
> 中间件是指从请求字符串和表单（如果请求是 url 编码的表单）中解析出 url 编码的参数。它向请求映射表中添加以下键。

:query-params：映射表，包含来自查询字符串的参数；

:form-params：映射表，包含来自请求体的参数；

:params：所有类型参数合并后的映射表。

它接受可选的配置映射表。识别的键包括：

:encoding：url 解码时用的编码方式。如果不指定，就用请求字符的编码，在没设置请求字符编码时用“UTF-8”。

没有只能使用一个中间件的限制。通常，你至少会用到 cookie、会话和参数中间件。有一种简洁的方式在处理函数中包装几个中间件，即使用 -> 宏：

```
(require '[ring.middleware.session :refer [wrap-session]])
(require '[ring.middleware.cookies :refer [wrap-cookies]])
(require '[ring.middleware.params :refer [wrap-params]])
(def wrapped-handler
  (-> handler
    wrap-cookies
    wrap-params
    wrap-session))
```

## 参阅

- Ring 的中间件概念文档（https://github.com/ring-clojure/ring/wiki/Concepts#middleware）。

# 7.3 用Ring提供静态文件

作者：Clinton Dreisbach

## 问题

希望通过 Ring 应用提供静态文件。

## 解决方案

使用 `ring.middleware.file/wrap-file`：

```
(require '[ring.middleware.file :refer [wrap-file]])

;; 提供公有目录下的所有文件
(def app
  (wrap-file handler "/var/webapps/public"))
```

## 讨论

`wrap-file` 包装了另一个 Web 请求处理函数，如果静态文件在指定的目录下存在，就提供

该文件，否则就调用该处理函数。

`wrap-file` 只是提供静态文件的一种方式。如果只想提供某一个文件，`ring.util.response/file-response` 将返回一个处理函数，提供该文件：

```
(require '[ring.util.response :refer [file-response]])

;; 提供 README.html
(file-response "README.html")

;; 提供 public/ 目录下的 README.html
(file-response "README.html" {:root "public"})

;; 通过符号连接提供 README.html
(file-response "README.html" {:allow-symlinks? true})
```

你常常希望将静态文件与应用捆绑在一起。在这种情况下，通过 classpath 而不是指定目录来提供文件就更有意义。要做到这一点，就用 `ring.middleware.resource/wrap-resource`：

```
(require '[ring.middleware.resource :refer [wrap-resource]])

(def app
  (wrap-resource handler "static"))
```

这将提供 classpath 中名为 static 目录下的所有文件。在 Leiningen 项目中，可以将 static 目录放在 resources/ 目录下，让静态文件和项目生成的 JAR 文件打包在一起。

也许你希望用 `ring.middleware.file-info/wrap-file-info` 包装所有的文件响应。这个 Ring 中间件检查文件的修改日期和类型，设置响应头中的 `Content-Type` 和 `Last-Modified`。`wrap-file-info` 需要包装 `wrap-file` 或 `wrap-resource`。

## 参阅

- 4.4 节“访问资源文件”。
- 7.8 节“用 Compojure 路由请求”。

# 7.4 用Ring处理表单数据

作者：Adam Bard

## 问题

希望应用能接受用户的 HTML 表单输入。

## 解决方案

利用 `ring.middleware.params/wrap-params`，将传入的 HTTP 表单参数添加到传入的请求映射表中。

要继续这个实例，请复制 https://github.com/clojure-cookbook/ringtest 代码库，并覆写 src/ringtest.clj：

```
(ns ringtest
  (:require
    [ring.adapter.jetty :as jetty]
    [ring.middleware.params :refer [wrap-params]]))

(def greeting-form
  (str
    "<html>"
    "  <form action='' method='post'>"
    "    Enter your name: <input type='text' name='name'><br/>"
    "    <input type='submit' value='Say Hello'>"
    "  </form>"
    "</html>"))

(defn show-form []
  {:body greeting-form
   :status 200 })

(defn show-name
  "A response showing that we know the user's name"
  [name]
  {:body (str "Hello, " name)
   :status 200})

(defn handler
  "Show a form requesting the user's name, or greet them if they
  submitted the form"
  [req]
  (let [name (get-in req [:params "name"])]
    (if name
      (show-name name)
      (show-form))))

(defn -main []
  ;; 在端口 3000 运行服务器
  (jetty/run-jetty (wrap-params handler) {:port 3000}))
```

## 讨论

`wrap-params` 是一个 Ring 中间件，它负责从原始请求中提取查询字符串和表单参数。它向请求中添加三个键。

`:query-params`

包含解析过的查询字符串参数的映射表。

`:form-params`

包含表单体参数的映射表。

`:params`

包含 `:query-params` 和 `:form-params` 合并后的内容。

在前面的例子中，我们使用了 `:form-params`，所以处理函数只响应 POST 请求，而且此请求包含以表单形式编码的参数。如果我们用的是 `:params`，就可以选择用 URL 查询字符串带上 `"name"` 参数。`:params` 适用于所有参数（表单形式或 URL 编码形式）。`:form-params` 只适合表单参数。

请注意，表单键被作为字符串传入，而不是关键字。

## 参阅

- 7.2 节“使用 Ring 中间件”。
- Ring 的参数文档（https://github.com/ring-clojure/ring/wiki/Parameters）。

# 7.5 用Ring处理Cookie

作者：Adam Bard

## 问题

你的 Web 应用需要读取或设置用户浏览器的 cookie（例如，为了记住用户的姓名）。

## 解决方案

用 `ring.middleware.cookies/wrap-cookies` 中间件，在请求中添加 cookie。

要继续这个实例，请复制 https://github.com/clojure-cookbook/ringtest 代码库，并覆写 src/ringtest.clj：

```
(ns ringtest
  (:require
    [ring.adapter.jetty :as jetty]
    [ring.middleware.cookies :refer [wrap-cookies]]
    [ring.middleware.params :refer [wrap-params]]))

(defn set-name-form
```

```
  "A response showing a form for the user to enter their name."
  []
  {:body "<html>
           <form action=''>
             Name: <input type='text' name='name'>
             <input type='submit'>
           </form>
         </html>"
   :status 200
   :content-type "text/html"})

(defn show-name
  "A response showing that we know the user's name"
  [name]
  {:body (str "Hello, " name)
   :cookies {"name" {:value name}} ; 保存 cookie
   :status 200 })

(defn handler
  "If we know the user's name, show it; else, show a form to get it."
  [req]
  (let [name (or
              (get-in req [:cookies "name" :value])
              (get-in req [:params "name"]))]
    (if name
      (show-name name)
      (set-name-form))))

(def wrapped-handler
  (-> handler
      wrap-cookies
      wrap-params))

(defn -main []
  ;; 在端口 3000 运行服务器
  (jetty/run-jetty wrapped-handler {:port 3000}))
```

# 讨论

这个例子使用了 `ring-core` 中包含的 `wrap-cookies` 和 `wrap-params` 中间件。用户第一次访问一个页面时，它显示一个表单，让他们输入姓名。一旦输入过，它就将用户的姓名保存在一个 cookie 中，并显示它们，直到该 cookie 被删除为止。

这个例子利用 `wrap-cookies`，从 cookie 映射表中取得用户保存的姓名，如果没有，就利用 `wrap-params`，从请求参数中取得用户的姓名。

Ring 的 cookie 中间件只在传入的请求映射表中，添加了一个额外的参数 `:cookies`，并将响应中传出的 cookie 设置为 `cookies` 参数。映射表中的 `:cookies` 参数看起来像这样：

```
{"name" {:value "Some Guy"}}
```

你可以为每个 cookie 添加其他可选的参数，和 `:value` 放在一起。Ring cookie 文档（https://github.com/ring-clojure/ring/wiki/Cookies）中说。

在设置 cookie 值的同时，你也可以设置其他属性。

- `:domain`—限制 cookie 仅用于特定的域名。
- `:path`—限制 cookie 仅用于特定的路径。
- `:secure`—如果为真，限制 cookie 仅用于 HTTPS URL。
- `:http-only`—如果为真，限制 cookie 仅用于 HTTP（比如不能通过 JavaScript 访问）。
- `:max-age`—cookie 过期的秒数。
- `:expires`—cookie 过期的指定日期和时间。

## 参阅

- 7.2 节“使用 Ring 中间件”。
- Ring 的 cookie 文档（https://github.com/ring-clojure/ring/wiki/Cookies）。

# 7.6 用Ring保存会话

作者：Adam Bard

## 问题

需要保存登录用户的一些安全数据，作为服务器端的状态。

## 解决方案

用 `ring.middleware.session/wrap-session`，在 Ring 应用中添加会话。

要继续这个实例，请复制 https://github.com/clojure-cookbook/ringtest 代码库，并覆写 src/ringtest.clj：

```
(ns ringtest
  (:require
    [ring.adapter.jetty :as jetty]
    [ring.middleware.session :refer [wrap-session]]
    [ring.middleware.params :refer [wrap-params]]))

(def login-form
  (str
    "<html>"
    "  <form action='' method='post'>"
```

```
    "   Username: <input type='text' name='username'><br/>"
    "   Password: <input type='text' name='password'><br/>"
    "   <input type='submit' value='Log In'>"
    "  </form>"
    "</html>"))

(defn show-form []
  {:body login-form
   :status 200 })

(defn show-name
  "A response showing that we know the user's name"
  [name session]
  {:body (str "Hello, " name)
   :status 200
   :session session })

(defn do-login
  "Check the submitted form data and update the session if necessary"
  [params session]
  (if (and (= (params "username") "jim")
           (= (params "password") "password"))
    (assoc session :user "jim")
    session))


(defn handler
  "Log a user in, or not"
  [{session :session params :form-params :as req}]
  (let [session (do-login params session)
        username (:user session)]

    (if username
      (show-name username session)
      (show-form))))

(def wrapped-handler
  (-> handler
      wrap-session

      wrap-params))

(defn -main []
  ;; 在端口 3000 运行服务器
  (jetty/run-jetty wrapped-handler {:port 3000}))
```

## 讨论

Ring 的会话中间件（https://github.com/ring-clojure/ring/wiki/Sessions）具有和 cookie API 类似的 API。你从 `:session` 请求映射表中取得会话数据，在响应映射表中加入 `:session` 键来设置它。在 `:session` 中写些什么取决于你，但通常会用一个映射表来保存一些键和值。

在背后，Ring 将设置名为 `ring-session` 的 cookie，它包含一个标识会话的唯一 ID。在请

求到达时，会话中间件从请求中取得会话 ID，然后从某个会话存储库中读取会话的值。

中间件使用哪个会话存储库是可以配置的。默认是使用内存中的会话存储库，这对于开发很有用，但副作用是重启应用时会丢失会话。Ring 还包括一个加密的 cookie 存储库，它是持久的。你还可以得到针对许多流行存储库编写的第三方库，包括 Memcached（https://github.com/killme2008/ring-session-memcached） 和 Redis（https://github.com/wuzhe/clj-redis-session）。你也可以编写自己的代码，将会话保存在任何数据库中。

向 `wrap-session` 传入一个选项映射表，其中包含 `:store` 参数，可以设置自己的存储库：

```
(wrap-session handler {:store (my-store)}))
```

要设置 `:session` 的值，只要将它和响应一起传出。如果不需要改变会话，就不要在响应中带上 `:session`。如果确实想清除会话，就将 `:session` 键的值设置为 `nil`。

## 参阅

- 7.2 节“使用 Ring 中间件”。
- Ring 的会话文档（https://github.com/ring-clojure/ring/wiki/Sessions）。

# 7.7 在Ring中读写请求和响应的头

作者：Luke VanderHart 和 Adam Bard

## 问题

需要读写 HTTP 请求和响应的头。

## 解决方案

从 Ring 请求映射表中读取 `:headers` 键，或在 Ring 处理函数返回之前，利用 `assoc` 将值关联到响应映射表中。

要继续这个实例，请复制 https://github.com/clojure-cookbook/ringtest 代码库，并覆写 src/ringtest.clj：

```
(ns ringtest
  (:require
    [ring.adapter.jetty :as jetty]))

(defn user-agent-as-json
  "A handler that returns the User-Agent header as a JSON
   response with an appropriate Content-Type"
  [req]
```

```
  {:body (str "{\"user-agent\": \"" (get-in req [:headers "user-agent"]) "\"}")
   :headers {"Content-Type" "application/json"}
   :status 200})

(defn -main []
  ;; 在端口 3000 运行服务器
  (jetty/run-jetty user-agent-as-json {:port 3000}))
```

## 讨论

这个例子定义了一个 Ring 处理函数，它将请求中的 `User-Agent` 头以 JSON 格式写入响应。它从请求头映射表中取得 `User-Agent`，利用响应中的 `Content-Type` 头来告诉客户端，响应应该解析为 JSON。

Ring 将请求头作为请求映射表中的 `:headers` 参数传入，也接受响应映射表中的 `:headers` 参数。头映射表中的键和值都应该是字符串。Clojure 关键字是不支持的。

可以利用 Ring 来设计所有 HTTP（http://en.wikipedia.org/wiki/List_of_HTTP_header_fields）中合法的头。

根据 RFC-2616（http://www.w3.org/Protocols/rfc2616/rfc 2616.html），头的名称不区分大小写。为了更容易一致地用 `get` 取得请求映射表中的值，不论大小写如何，Ring 都以小写传入所有头的值，也不论来自什么客户端。但是，你可能希望在发送头时使用规范的大小写，以防面对的客户端不兼容（遵循经典的健壮性原则："对发送的保守，对接收的开放"）。

## 参阅

- Ring 的概念文档（https://github.com/ring-clojure/ring/wiki/Concepts）。

# 7.8　用Compojure路由请求

作者：Adam Bard

## 问题

需要一种简单的方式将 URL 路由到特定的 Ring 处理函数。

## 解决方案

用 Compojure 库（https://github.com/weavejester/compojure）为应用添加路由。

要继续这个实例，请复制 https://github.com/clojure-cookbook/ringtest 代码库，并覆写 src/ringtest.clj：

```
(ns ringtest
  (:require
    [compojure.core :refer [defroutes GET]]
    [ring.adapter.jetty :as jetty]))

;; 一些视图函数
(defn view [x]
  (str "<h1>" x "</h1>"))

(defn index []
  (view "Hello"))

(defn index-fr []
  (view "Bonjour"))

;; 路由
(defroutes main-routes
  (GET "/" [] (index))
  (GET "/en/" [] (index))
  (GET "/fr/" [] (index-fr))
  (GET "/:greeting/" [greeting] (view greeting)))

;; 服务器
(defn -main []
  (jetty/run-jetty main-routes {:port 3000}))
```

## 讨论

Compojure 是一个路由库，让你在应用中定义路由。它是通过 `defroutes` 宏来完成的，该宏生成一个 Ring 处理函数。

这里，我们定义了四个路由：

`/`
: 显示 "Hello" ；

`/en/`
: 也显示 "Hello" ；

`/fr/`
: 显示 "Bonjour" ；

`/:greeting/`
: 回显用户传入的问候。

最后一个视图是 Compojure URL 参数语法的例子。URL 中由 `:greeting` 标识的部分被传递给视图，视图将它显示给用户。所以，访问 http://localhost:3000/Buenos%20Dias/ 将在响应中显示“Buenos Dias”。

有一点需要注意，Compojure 路由对于末尾的斜杠是敏感的：定义为 `/users/:user/blog/` 的路由不会匹配 URL http://mysite.com/users/fred/blog，而是匹配 http://mysite.com/users/fred/blog/。

每个路由中的 [] 实际上是语法糖，用于截获这些参数。也可以用 `req` 或其他任何符号来取得完整的请求：

```
(defroutes main-routes-2
  (GET "/" req (some-view req)))
```

你甚至可以用 Clojure 的解构语法[1]，提取请求中的部分。例如，如果使用 `wrap-params` 中间件，就可以抓取参数并将它们传递给一个函数：

```
(defroutes main-routes-2
  (GET "/" {params :params} (some-other-view params)))
```

重要的是要意识到，Compojure 是基于 Ring 的。你的视图仍然应该返回 Ring 的响应映射表（尽管 Compojure 会对基本的 200 响应返回的东西进行包装）。

也可以为其他类型的 HTTP 请求定义路由，只要用相关的 Compojure 指令来指定：除了 `compojure.core/GET`，`compojure.core/POST` 和 `compojure.core/PUT` 也是最常用的。

Compojure 还提供了其他一些有用的工具，如 `compojure.route`（http://weavejester.github.io/compojure/compojure.route.html），包含了辅助函数来提供资源、文件和 404 响应，又如 `compojure.handler`（http://weavejester.github.io/compojure/compojure.handler.html），将一些 Ring 中间件打包成一个方便的包装函数。

## 参阅

- Compojure 的网站（http://compojure.org/）。

# 7.9 用Ring执行HTTP重定向

作者：Craig McDaniel

## 问题

在 Ring 应用中，希望返回 HTTP 响应码，将浏览器重定向到另一个 URL。

注 1：如果你不熟悉 Clojure 的解构语法，建议阅读 Jay Fields 的“Clojure: Destructuring”博客文章（http://blog.jayfields.com/2010/07/clojure-destructuring.html）。更多的资源，请阅读 *Clojure Programming*（http://www.clojurebook.com/, O’Reilly），作者是 Chas Emerick、Brian Carper 和 Christophe Grand。该书深入介绍了解构。

## 解决方案

要重定向 Ring 请求，可以使用 `ring.util.response` 命名空间的 `redirect` 函数。

要继续这个实例，请复制 https://github.com/clojure-cookbook/ringtest 代码库，并覆写 src/ringtest.clj：

```
(ns ringtest
  (:require
   [ring.adapter.jetty :as jetty]
   [ring.util.response :as response]))

(defn redirect-to-github
  "A handler that redirects all requests"
  [req]
  (response/redirect "http://github.com/"))

(defn -main []
  ;; 在端口 3000 运行服务器
  (jetty/run-jetty redirect-to-github {:port 3000}))
```

## 讨论

`ring.util.response` 命名空间包含了一个重定向 URL 的函数。这个 URL 可以从请求映射表中动态生成（利用来自 `wrap-params` 的参数、请求头等）。在背后，这个函数只是创建了一个响应映射表，包含 `:status` 值为 302 以及一个位置头，指明要重定向到的 URL。

根据 HTTP 规范，如果响应方法是 POST、PUT 或 DELETE，就应该假定服务器接受了请求，客户端应该向位置头中的 URL 发起 GET 请求。在编写 REST 服务时，这是一项重要的警告。幸运的是，规范提供了 307 状态码，它将指示客户端，请求应该重定向到新的位置，并使用原来的方法和请求体。要做到这一点，只要在处理函数中像这样返回响应映射表：

```
(defn redirect-to-github
  [req]
  {:status 307
   :headers {"Location" "http://github.com"}
   :body ""})
```

## 参阅

- Ring 的概念文档（https://github.com/ring-clojure/ring/wiki/Concepts）。

# 7.10　用Liberator构建REST风格的应用

作者：Eric Normand

## 问题

希望基于 Ring 和 Compojure，在较高的抽象层上构建 REST 风格（符合 RFC 2616）的 Web 应用，即定义资源。

## 解决方案

使用 Liberator（https://github.com/clojure-liberator/liberator）来创建符合 HTTP 的、REST 风格的 Web 应用。

要继续这个实例，请用 **lein new liberator-test** 命令创建一个新项目。

在 project.clj 中，将下面的依赖关系添加到 `:dependencies` 键：

```
[compojure "1.0.2"]
[ring/ring-jetty-adapter "1.1.0"]
[liberator "0.9.0"]
```

然后将 src/liberator_test/core.clj 修改成下面的内容：

```
(ns liberator-test.core
  (:require [compojure.core :refer [defroutes ANY]]
            [ring.adapter.jetty :as jetty]
            [liberator.core :refer [defresource]]))

;; 资源

(defresource root
   :available-media-types #{"text/plain"})

;; 路由
(defroutes main-routes
  (ANY "/" [] root))

;; 服务器
(defn -main []
 (jetty/run-jetty main-routes {:port 3000}))
```

## 讨论

Liberator（https://github.com/clojure-liberator/liberator）是一个库，用于开发符合 HTTP 的 Web 服务器。它处理 REST 风格资源的内容交互、状态码和标准请求方法。它利用一棵符合 HTTP 规范的决策树来决定响应的状态码。

Liberator 不处理路由，所以需要用另一个库。这个实例使用了 Compojure。由于 Liberator 在处理请求方法（GET、PUT、POST 等）时更好，所以应该在 Compojure 路由中使用 `ANY`。也可以用不同的路由库，如 Clout（https://github.com/weavejester/clout）、Moustache

（https://github.com/cgrand/moustache）或 playnice（https://github.com/ericnormand/playnice）router。

`defresource` 格式定义了一个 Web 资源，它被建模为一个 Ring 处理函数。然后你可以将该资源作为最后的参数，传递给 Compojure 路由。

Liberator 资源建立时具有合理的默认值。可用媒质类型的默认值是空集，所以需要设置。否则，Liberator 将返回 406“Not Acceptable”的响应。在这个实例中，它被设置为响应时带有 `text/plain` 的 MIME 类型。默认的响应是“OK”，如果你运行本实例，并用浏览器访问 http://localhost:3000，就会看见。

## 参阅

- 7.1 节“Ring 简介”，了解关于设置 Ring 的更多信息。
- 7.8 节“用 Compojure 路由请求”，了解关于 Compojure 路由的更多信息。
- Liberator 的主页（http://clojure-liberator.github.io/liberator/）。

# 7.11 用Enlive实现HTML模板

作者：Luke VanderHart

## 问题

希望基于模板动态地创建 HTML，不用传统的混合代码，或 DSL 风格的模板。

## 解决方案

使用 Enlive（https://github.com/cgrand/enlive），它是一个 Clojure 库，采用基于选择器的方式来实现 HTML 模板。

不像 PHP、ERB 和 JSP 等其他模板框架，它不混合代码和文本。不像 Haml 或 Hiccup 这样的系统，它不使用特殊的 DSL。相反，模板是普通的 HTML 文件，Enlive 利用 Clojure 代码来定位具体的区域，并用传入的数据实现替代或复制。

要继续本实例，请用 `lein-try` 启动 REPL：

```
$ lein try enlive
```

开始，创建 post.html 文件，作为 Enlive 模板：

```
<html>
  <head><title>Page Title</title></head>
  <body>
    <h1>Page Title</h1>
```

```
    <h3>By <span class="author">Mickey Mouse</span></h3>
    <div class="post-body">
      Lorem ipsum etc...
    </div>
  </body>
</html>
```

如果你准备在项目中使用 Enlive，就将这个文件放在 resources/ 目录中。

下面的 Clojure 代码根据 post.html 的内容，定义了一个 Enlive 模板：

```
(require '[net.cgrand.enlive-html :as html])

;; 定义模板
(html/deftemplate post-page "post.html"
  [post]
  [:title] (html/content (:title post))
  [:h1] (html/content (:title post))
  [:span.author] (html/content (:author post))
  [:div.post-body] (html/content (:body post)))

;; 一些示例数据
(def sample-post {:author "Luke VanderHart"
                  :title "Why Clojure Rocks"
                  :body "Functional programming!"})
```

要对数据应用该模板，就调用 `deftemplate` 定义的函数。因为它返回一个字符串序列，所以在大多数应用中，你会希望将结果连接成一个字符串：

```
(reduce str (post-page sample-post))
```

下面是格式化的输出：

```
<html>
  <head><title>Why Clojure Rocks</title></head>
  <body>
    <h1>Why Clojure Rocks</h1>
    <h3>By <span class="author">Luke VanderHart</span></h3>
    </h3><div class="post-body">Functional programming!</div>
  </body>
</html>
```

关于 `deftemplate` 宏的详细解释，以及这段代码究竟做了些什么，请参见后面的讨论小节。

### 重复元素

前面的代码只是在输出的 HTML 中，取代了特定节点的值。在真实的场景中，另一个常见的任务是重复输入的 HTML 中的某些项，每次重复都展示一项输入数据。对于这个任务，

Enlive 提供了片断（snippet），它是输入的 HTML 的节选，可以在另一个模板的输出中被重复任意次：

```
(def sample-post-list
  [{:author "Luke VanderHart"
    :title "Why Clojure Rocks"
    :body "Functional programming!"}
   {:author "Ryan Neufeld"
    :title "Clojure Community Management"
    :body "Programmers are like..."}
   {:author "Rich Hickey"
    :title "Programming"
    :body "You're doing it completely wrong."}])

(html/defsnippet post-snippet "post.html"
  {[:h1] [[:div.post-body (html/nth-of-type 1)]]}
  [post]
  [:h1] (html/content (:title post))
  [:span.author] (html/content (:author post))
  [:div.post-body] (html/content (:body post)))

(html/deftemplate all-posts-page "post.html"
  [post-list]
  [:title] (html/content "All Posts")
  [:body] (html/content (map post-snippet post-list)))
```

调用定义的 `all-posts-page` 函数将得到一个 HTML 页面，其中填充了全部三个海报：

```
(reduce str (all-posts-page sample-post-list))
```

下面是格式化后的输出：

```
<html>
  <head><title>All Posts</title></head>
  <body>
    <h1>Why Clojure Rocks</h1>
    <h3>By <span class="author">Luke VanderHart</span></h3>
    <div class="post-body">Functional programming!</div>
    <h1>Clojure Community Management</h1>
    <h3>By <span class="author">Ryan Neufeld</span></h3>
    <div class="post-body">Programmers are like...</div>
    <h1>Programming</h1>
    <h3>By <span class="author">Rich Hickey</span></h3>
    <div class="post-body">You're doing it completely wrong.</div>
  </body>
</html>
```

在这个例子中，`defsnippet` 宏定义了一个片段，包含了输入 HTML 中的一些元素，从 `<h1>` 到 `<div class="post-body">`。然后，`all-posts-page` 的 `deftemplate` 使用了 `post-snippet` 映射到 `body` 元素的内容之后的结果。由于示例输入数据中有三个海报，这个片段被求值三次，得到的 HTML 包含了三个海报的输出。

## 讨论

与其他一些库相比，Enlive 可能有一点难上手。这有以下几个原因。

- 它比其他模板系统具有更创新的概念方法（尽管它与其他一些非 Clojure 模板技术有许多相似之处，如 XSLT）。
- 它充分利用了函数式编程技术，包括自由使用高阶函数。
- 它是一个很大的库，能做很多事。完成某个任务所需的功能子集不一定很明显。

一般来说，要解决这些问题并体验 Enlive 的强大和灵活，最好的方法就是分别理解所有不同的部分，以及它们做些什么。然后，将它们组合成有用的模板系统，变得更容易管理。

### Enlive和DOM

首先，要理解 Enlive 不直接操作 HTML 文本，这一点很重要。相反，它先将 HTML 解析成 Clojure 数据结构，表示为 DOM（文档对象模型）。例如，HTML 片段：

```
<div id="foo">
  <span class="bar">Hello!</span>
</div>
```

将被解析为下面的 Clojure 数据：

```
{:tag :html,
  :attrs nil,
  :content
  ({:tag :body,
    :attrs nil,
    :content
    ({:tag :div,
      :attrs {:id "foo"},
      :content
      ({:tag :span, :attrs {:class "bar"}, :content ("Hello!")})})})}
```

这更冗长，但更容易在 Clojure 中操作。你不需要直接处理这些数据结构，但要意识到，Enlive 说它操作一个元素或节点时，是指该元素的 Clojure 数据结构，而不是 HTML 字符串。

### 模板

这些例子中更重要的部分是 `deftemplate` 宏。`deftemplate` 接受一个符号作为名称，一个类路径的相对路径指向一个 HTML 文件，一个参数列表，一系列的*选择器*（selector）和*转换函数*（transform function）对。它定义了一个函数，具有同样名称和指定的参数。调用这个函数，就返回一个字符串序列，作为产生的 HTML。

Enlive 的*选择器*是一个 Clojure 数据结构，确定了输入 HTML 文件中的某个节点。它们在操作上与 CSS 选择器类似，但能力更强。在解决方案的例子中，`[:title]` 选择了每个

<title>，[:span.author] 选择了每个带有 class="author" 的 <span>，等等。下面的小节描述了更多的选择器形式。

模板的*转换函数*接受一个 Enlive 节点，返回修改后的节点。我们的例子使用了 Enlive 的 content 工具函数，它返回一个函数，该函数用它的参数值替换节点的内容。

返回值本身不是一个字符串，而是一个字符串序列，每个字符串表示一小段 HTML 内容。这使得底层的数据结构能够惰性地转换为字符串的形式。为了简单，我们的例子对结果归约调用了 str 来连接字符串，但这样做实际上不能达到最佳性能。要高效地生成一个字符串，请用 Java 的 StringBuilder 类，它利用了可变的状态来构建 String 对象，性能最好。或者，完全跳过字符串合并，将模板函数产生的序列直接传给输出的 Writer 对象，大多数 Web 应用库（包括 Ring）都可以用它作为 HTTP 的响应体（模板化的 HTML 最常见的去处）。

### 选择器

Enlive 选择器是一些数据结构，确定一个或多个 HTML 节点。它们描述了一种数据*模式*，如果该模式匹配 HTML 数据结构中的某些节点，选择器就会选择这些节点。选择器可能从给定的 HTML 文档中选择一个、多个或零个节点，这取决于该模式有多少匹配。

有效选择器形式的完整参考相当复杂，也超出了本实例的范围。完整的文档请参考选择器的规格说明（http://enlive.cgrand.net/syntax.html）。下面的选择器模式应该足够让你开始工作了：

[:div]
: 选择所有 <div> 元素节点。

[:div.sidebar]
: 选择所有带有 CSS 类 "sidebar" 的 <div> 元素节点。

[:div#summary]
: 选择带有 HTML ID "summary" 的 <div> 元素节点。

[:p :span]
: 选择所有 <p> 元素下面的 <span> 元素。

[:div.menu :ul :li :span]
: 选择 <div> 元素之内的 <ul> 元素之内的 <li> 元素之内的 <span> 元素，<div> 元素带有的 CSS 风格为 "menu"。

[[:div (nth-child 2)]]
: 选择所有是父元素的第二个子元素的 <div> 元素。双方括号不是打字错误，因为内部

的向量表示一种逻辑“与”条件。在这个例子中，匹配的元素必须是 `<div>`，并且 `nth-child` 谓词必须为真。

除 `nth-child` 外，还有其他谓词，也可以定义自己的谓词。详细内容请参见 Enlive 文档。

最后，有一种特殊类型的选择器名为范围（range）选择器，它不是由一个向量来指定，而是由一个映射表字面量（在花括号内）来指定。范围选择器包含另两个选择器，它按照文档的顺序，匹配两个匹配节点之间的所有节点（含这两个节点）。开始节点在映射表常量中处于键的位置，结束节点处于值的位置，所以选择器 `{[:.foo] [:.bar]}` 将匹配 ID “foo”和 ID “bar”之间的所有节点。

解决方案中的例子在 `defsnippet` 形式中使用了一个范围选择器，来选择逻辑上同一篇博客文章的所有节点，即使它们没有嵌套在同一个父元素中。

### 片段

片断与模板类似，它们都基于 HTML 文件得到一个函数。但是，片断与模板有两个主要的不同点。

(1) 不像模板那样总是生成整个 HTML 文件，片断只渲染输入 HTML 的一部分。要渲染的部分是由 Enlive 选择器指定的，该选择器作为 `defsnippet` 宏的第三个参数，跟在名称和 HTML 文件路径之后。

(2) 生成函数的返回值是 Enlive 数据结构，不是 HTML 字符串。这意味着渲染一个片段的结果可以直接来自一个模板的转换函数或其他片段的结果。这就是 Enlive 开始展示其强大的地方，片段能被回收和大量复用，用于不同的组合。

除了这些不同，`defsnippet` 形式和 `deftemplate` 一样，在选择器之后，剩下的参数也一样：一个参数向量，一系列的选择器和转换函数对。

### 用Enlive提取数据

由于它强调选择器，而且使用了普通的、无标注的 HTML 文件，所以 Enlive 不仅非常适合利用模板来生成 HTML，也非常适合解析来自任何来源的 HTML，并从中提取数据。

要利用 Enlive 从 HTML 中提取数据，必须先将 HTML 文件解析成 Enlive 数据结构。要做到这一点，只要对 HTML 文件调用 `net.cgrand.enlive-html/html-resource` 函数。你可以将该文件指定为一个 `java.net.URL` 对象、一个 `java.io.File` 对象，或一个表示类路径的相对路径的字符串。函数将返回解析后的 Enlive 数据结构，表示 HTML 的 DOM。

然后，可以用 `net.cgrand.enlive-html/select` 函数，将选择器应用于 DOM 并提取特定的数据。给定一个节点和一个选择器，`select` 将只返回那些匹配的节点。接下来可以用 `net.cgrand.enlive-html/text` 函数取得节点的文本内容。

例如，下面的函数将返回一个序列，包含 XKCD 档案中最近 n 个漫画书标题：

```
(defn comic-titles
  [n]
  (let [dom (html/html-resource
              (java.net.URL. "http://xkcd.com/archive"))
        title-nodes (html/select dom [:#middleContainer :a])
        titles (map html/text title-nodes)]
    (take n titles)))

(comic-titles 5)
;; -> ("Oort Cloud" "Git Commit" "New Study"
       "Telescope Names" "Job Interview")
```

### 何时使用Enlive

作为 HTML 模板系统，Enlive 有两个主要的价值点，胜过 Clojure 生态系统中的其他可选方案。

首先，模板是纯 HTML。这使得与 HTML 设计师合作变得比较容易：他们可以将 HTML 的设计直接交给开发者，不必在里面嵌入标记代码，开发者可以直接使用，不必手工进行切割（也就是说，在代码之外）。而且，模板本身可以在浏览器中静态地查看，这意味着它们可以作为自己的页面线框图（wireframe）。这消除了负担，不需要保持 Web 项目的视觉原型与代码的同步。

其次，因为它使用了真正的 Clojure 函数和数据结构，而不是定制的 DSL，所以 Enlive 充分展示了 Clojure 语言的威力。几乎很难发现 Enlive 的能力有什么限制，只要用 Clojure 的函数和宏就可以扩展它，操作熟悉的、稳定的、不可改变的数据结构。

## 参阅

- Enlive 的文档（https://github.com/cgrand/enlive/wiki）。
- David Nolen 的 Enlive 指南（https://github.com/swannodette/enlive-tutorial）。
- Enlive 的邮件列表（https://groups.google.com/forum/#!forum/enlive-clj）。
- 另外的模板库 Selmer（7.12 节）和 Hiccup（7.13 节）。

# 7.12 用Selmer实现模板

作者：Dmitri Sotnikov

## 问题

希望使用类似 Django 和 Jinja 的语法，创建服务器端的页面模板。希望能够插入动态内容，并使用模板继承来构成模板。

## 解决方案

使用 Selmer 库（https://github.com/yogthos/Selmer）来创建模板，并用一个包含动态内容的上下文映射表来调用它。

要继续本实例，请用 `lein-try` 启动 REPL：

```
$ lein try selmer
```

Selmer 模板是包含特殊标签的 HTML 文档，这些标签将在运行时被动态内容填充。一个简单的模板（base.html）看起来是这样的：

```
<!DOCTYPE html>
<html lang="en">
  <body>
    <header>

      <h1>{{header}}</h1>

      <ul id="navigation">

        {% for item in nav-items %}
        <li>
            <a href="{{item.link}}">{{item.name}}</a>
        </li>
        {% endfor %}

      </ul>
    </header>
  </body>
</html>
```

调用 `selmer.parser/render-file` 函数，可以渲染该模板：

```
(require '[selmer.parser :refer [render-file]])

(println
  (render-file "base.html"
               {:header "Hello Selmer"
                :nav-items [{:name "Home" :link "/"}
                            {:name "About" :link "/about"}]}))
```

在 `render-file` 执行时，它将用提供的内容填充那些标签。值将作为一个字符串返回，适合作为 Ring 应用的响应体（比方说）。这里只是简单地在标准输出中打印出结果，目的是易于查看。

在运行时，我们可以对变量应用过滤器，进行附加的事后处理。在这里，我们用 `upper` 过滤器将标题转换成大写：

```
<h1>{{header|upper}}</h1>
```

利用 include 标签，我们可以提取模板的一些部分，成为独立的片段。例如，如果希望在独立的文件 header.html 中定义标题：

```
<header>

  <h1>{{header}}</h1>

  <ul id="navigation">

    {% for item in nav-items %}
    <li>
        <a href="{{item.link}}">{{item.name}}</a>
    </li>
    {% endfor %}

  </ul>
</header>
```

然后就可以这样包含它：

```
<!DOCTYPE html>
<html lang="en">
  <body>

    {% include "header.html" %}

  </body>
</html>
```

在模板编译时，include 标签就会被它所指文件的内容替换。

在创建独立的页面时，我们也可以扩展基本模板。要做到这一点，我们先在基本模板中定义一个块。它将作为一个锚，让子模板来覆写：

```
<!DOCTYPE html>
<html lang="en">
  <body>

    {% include "header.html" %}

    {% block content %}
    {% endblock %}

  </body>
</html>
```

子模板通过 extends 标签来引用父模板，并为 content 块定义自己的内容：

```
{% extends "base.html" %}

{% block content %}
```

```
<h1>This is the home page of the site</h1>
<p>some exciting content follows</p>

{% endblock %}
```

## 讨论

Selmer 提供了一个强大而熟悉的模板工具，包含许多标签和过滤器，很容易实现许多常见的任务。它通过设计分离了视图逻辑和展现。Selmer 的性能也很好，因为它编译模板，确保只在服务请求时才对动态内容求值。

### Selmer的概念

Selmer 包含两种类型的元素：变量和标签。

变量用于在页面上渲染来自上下文映射表中的值。{{ 和 }} 用于表明变量的起始和终止。

在许多时候，你可能希望对变量的值进行事后处理。例如，希望将它转换成大写，变成复数，或将它解析为日期。变量过滤器（在下面的小节中描述）就是用于这个目的。

标签用于向模板中添加各种功能，诸如循环和条件。我们已经看到了 `for`、`include` 和 `extends` 标签的例子。这些标签用 {% 和 %} 来定义它们的内容。

默认的标签字符可能与客户端的框架冲突，例如 AngularJS。在这种情况下，我们可以通过向解析器传入一个映射表指定定制的标签，包含以下键：

```
:tag-open
:tag-close
:filter-open
:filter-close
:tag-second
:custom-tags
:custom-filters
```

如果我们想用 [ 和 ] 作为开始标签和关闭标签，可以这样调用 `render` 函数：

```
(render (str "[% for ele in foo %] "
             "{{I'm not a tag, but the next one is}} [{ele}] [%endfor%]")
        {:foo [1 2 3]}
        {:tag-open \[
         :tag-close \]})
```

`render` 函数与 `render-file` 功能一样，只是它以字符串的方式接受模板内容。

### 定义过滤器

Selmer 提供了丰富的过滤器，用于修饰动态内容。例如 `capitalize`、`pluralize`、`hash`、`length` 和 `sort` 等。

但是，如果需要定制库中没有的过滤器，那也很容易。例如，如果需要用 `markdown-clj` 库

（https://github.com/yogthos/markdown-clj）来解析 Markdown 并显示在页面上，我们可以编写下面的过滤器[2]：

```
(require '[markdown.core :refer [md-to-html-string]]
         '[selmer.filters/add-filter!])

(add-filter! :markdown md-to-html-string)
```

现在我们可以在模板中用这个过滤器来渲染 Markdown 的内容：

```
<h2>Blog Posts</h2>
<ul>
  {% for post in posts %}
    <li>{{post.title|markdown|safe}}</li>
{% endfor %}
</ul>
```

请注意，我们必须在 `markdown` 过滤器后面加上 `safe` 过滤器。这是因为 Selmer 默认会转义可变的内容。我们可以改变过滤器的定义，告诉它不需要转义，就像这样：

```
(add-filter! :markdown (fn [s] [:safe (md-to-html-string s)]))
```

### 定义标签

同样，除了库里已有的标签，我们也可以定义定制的标签。这是通过调用 `selmer.parser/add-tag!` 函数来完成的。

假定我们希望添加一个标签，将它的内容转换成大写：

```
(require '[selmer.parser :refer [add-tag!]])

(add-tag! :uppercase
          (fn [args context-map content]
            (.toUpperCase (get-in content [:uppercase :content])))
          :enduppercase)

(render "{% uppercase %}foo {{bar}} baz{% enduppercase %}" {:bar "injected"})
```

### 继承

我们已经看到了一些模板继承的例子。每个模板都可以扩展一个模板，并且可以在内容中包含多个模板。继承的模板还可以再继承。在这种情况下，最下层子模板中的块将覆写所有名称一样的块。

## 参阅

- Selmer 的 GitHub 代码库（https://github.com/yogthos/Selmer）。

---

注 2：要尝试这个例子，你需要用 `lein-try` 重新启动 REPL，带上 `markdown-clj`。

# 7.13 用Hiccup实现模板

作者：Ryan Neufeld

## 问题

希望基于模板动态地创建 HTML，该模板由纯 Clojure 数据构成。

## 解决方案

使用 Hiccup 库，它展现并渲染由普通 Clojure 数据结构构成的 HTML 模板。

要继续这个实例，就用 `lein-try` 启动 REPL：

```
$ lein try hiccup
```

Hiccup 将 HTML 节点表示为向量。向量的第一项是元素的名称，第二项是一个元素属性的可选映射表，剩下的项是元素体：

```
;; <h1 class="header">My Page Title</h1> in Hiccup...
[:h1 {:class "header"} "My Page Title"]

;; <ul>
;;   <li>lions</li>
;;   <li>tigers</li>
;;   <li>bears</li>
;; </ul> in Hiccup...
[:ul
  [:li "lions"]
  [:li "tigers"]
  [:li "bears"]] ;; oh my!
```

用 `hiccup.core/html` 函数将 Hiccup 数据结构渲染成 HTML：

```
(require '[hiccup.core :refer [html]])
(html [:h1 {:class "header"} "My Page Title"])
;; -> "<h1 class=\"header\">My Page Title</h1>"
```

由于节点表示为普通的 Clojure 数据，所以你可以用任何 Clojure 内建的函数或技术来生成符合 Hiccup 要求的向量：

```
(def pi 3.14)
(html [:p (str "Pi is approximately: " pi)])
;; -> "<p>Pi is approximately: 3.14</p>"

(html [:ul
        (for [animal ["lions" "tigers" "bears"]]
          [:li animal])])
;; -> "<ul><li>lions</li><li>tigers</li><li>bears</li></ul>"
```

利用前面所有的技术，可以创建一个简单的函数，动态地填充一个最小的博客页面的内容，其中只用到 Clojure 的函数和数据：

```
(defn blog-index
  "Render a blog's index as Hiccup data"
  [title author posts]
  [:html
      [:head
      [:title title]]
    [:body
      [:h1 title]
      [:h2 (str "By " author)]
      (for [post posts]
        [:article
          [:h3 (:title post)]
          [:p (:content post)]])]])

(-> (blog-index "My First Blog"
                "Ryan"
                [{:title "First post!" :content "I'm here!"}
                 {:title "Second post." :content "Yawn, bored."}])

    html)
```

格式化后的输出是：

```
<html>
  <head>
    <title>My First Blog</title>
  </head>
  <body>
    <h1>My First Blog</h1>
    <h2>By Ryan</h2>
    <article>
      <h3>First post!</h3>
      <p>I'm here!</p>
    </article>
    <article>
      <h3>Second post.</h3>
      <p>Yawn, bored.</p>
    </article>
  </body>
</html>"
```

# 讨论

Hiccup 是一种简单的、干脆利落的模板方式，通过原始的函数和数据来渲染 HTML。如果你没有时间来学习一种新的 DSL，或者你只喜欢在 Clojure 下工作，它就特别方便。

一个 HTML 节点在 Hiccup 中表示为一个向量，包含以下一些元素：

- 节点的名称，表示为关键字（例如 `:h1`、`:article` 或 `:body`）；

- 包含节点属性的可选的映射表，属性名称表示为关键字（例如 `{:href "/posts/"}` 或 `{:id "post-1" :class "post"}`）；
- 任意数目的其他节点或字符串的值，构成该节点的节点体。

用单个节点、片段或整个页面来调用 `hiccup.core/html`，就会将它们的内容渲染成 HTML。对于带有特殊字符、应该转义的内容，通过调用 `hiccup.core/h` 将它们包装起来：

```
(require '[hiccup.core :refer [h]])
(html [:a {:href (h "/post/my<crazy>url")}])
;; -> "<a href=\"/post/my&amp;lt;crazy&amp;gt;url\"></a>"
```

Hiccup 对渲染表单也有基本的支持。使用 `hiccup.form` 命名空间的 `form-to` 和一些其他辅助函数，来简化渲染表单标签：

```
(require '[hiccup.form :as f])

(f/form-to [:post "/posts/new"]
  (f/hidden-field :user-id 42)
  (f/text-field :title)
  (f/text-field :content))
;; -> [:form {:method "POST", :action #<URI /posts/new>}
;;      [:input {:type "hidden"
;;               :name "user-id"
;;               :id "user-id"
;;               :value 42}]
;;      [:input {:type "text"
;;               :name "title"
;;               :id "title"
;;               :value nil}]
;;      [:input {:type "text"
;;               :name "content"
;;               :id "content"
;;               :value nil}]]
```

## 参阅

- Hiccup 的 GitHub 代码库（https://github.com/weavejester/hiccup/），API 文档（http://weavejester.github.io/hiccup/）和维基页面（https://github.com/weavejester/hiccup/wiki）。
- 如果对模板引擎有更复杂的需求（例如消费并填充已有的 HTML 文件），你可能需要像 Enlive（7.11 节）或 Selmer（7.12 节）这种更强大的工具。

# 7.14　渲染Markdown文档

作者：Dmitri Sotnikov

## 问题

需要渲染 Markdown 文档。

## 解决方案

使用 `markdown-clj` 库（https://github.com/yogthos/markdown-clj）来渲染 Markdown 文档。

要继续这个实例，请用 `lein-try` 启动 REPL：

```
$ lein try markdown-clj
```

用 `markdown.core/md-to-html` 来读取 Markdown 文档，并生成包含 HTML 的字符串：

```
(require '[markdown.core :as md])

(md/md-to-html "input.md" "output.html")

(md/md-to-html (input-stream "input.md") (output-stream "test.txt"))
```

用 `markdown.core/md-to-html-string` 将包含 Markdown 内容的字符串转换成 HTML 表现形式：

```
(md/md-to-html-string
  "# This is a test\n\nsome code follows:\n```\n(defn foo [])\n```")

<h1> This is a test</h1><p>some code follows:</p><pre>
&#40;defn foo &#91;&#93;&#41;
</pre>
```

## 讨论

Markdown 是一种流行的轻量级标记语言，很容易读写，并能转换成结构上有效的 HTML。

由于 Markdown 将渲染 HTML 的许多方面留给了解析器自由实现，所以不能保证不同的解析器会得到同样的结果。如果在客户端用一个解析器来渲染 Markdown 的预览图，然后在服务器端用另一个解析器来生成 HTML，这可能会有问题。由于 `markdown-clj` 被编译成 Clojure 和 ClojureScript 两个版本，所以它避免了这个问题。有了它，你可以在服务器和客户端使用同一个解析器，保证文档得到一致的渲染。

让我们来看看更多使用该库的例子。代码块可以用标注说明是哪种语言。在这个例子中，`pre` 标签将被一个类修饰，该类符合 SyntaxHighlighter（语法高亮器，http://alexgorbatchev.com/SyntaxHighlighter/）：

```
(md/md-to-html-string (str "# This is a test\n\nsome code follows:\n"
                           "```clojure\n(defn foo [])\n```"))

<h1> This is a test</h1><p>some code follows:</p><pre class="brush: clojure">
&#40;defn foo &#91;&#93;&#41;
</pre>
```

markdown-clj 支持所有标准的 Markdown 标签，但引用风格的链接除外（因为解析器用一遍的方式来生成文档）。markdown.core/md-to-html 一行一行地处理输入，在处理时，完整的内容不需要保存在内存中。另一方面，md-to-html-string 和 md-to-html 都会将全部内容加载到内存中。

解析器接受额外的格式化选项。这包括 :heading-anchors、:code-style、:custom-transformers 和 :replacement-transformers。

如果 :heading-anchors 键设置为 true，就会为每个标题标签生成一个锚：

```
(md/md-to-html-string "###foo bar BAz" :heading-anchors true)

<h3>
  <a name=\"heading\" class=\"anchor\" href=\"#foo&#95;bar&#95;baz></a>
  foo bar BAz
</h3>
```

:code-style 键允许覆写代码块的默认风格提示：

```
(md/md-to-html-string "```clojure\n(defn foo [])\n```"
                      :code-style #(str "class=\"" % "\""))

<pre class="clojure">
&#40;defn foo &#91;&#93;&#41;
</pre>
```

我们可以用 :custom-transformers 键，为定制的标签指定转换器。转换器函数应该接受 text 参数，它代表当前行，以及 state 参数，它包含解析器的当前状态。状态可以用来保存一些信息，比如哪些标签是激活的：

```
(defn capitalize [text state]
  [(.toUpperCase text) state])

(md/md-to-html-string "#foo" :custom-transformers [capitalize])

<H1>FOO</H1>
```

最后，我们可以用 :replacement-transformers 键提供一组定制的转换器，替代内建的转换器：

```
(markdown/md-to-html-string "#foo" :replacement-transformers [capitalize])
```

## 参阅

- markdown-clj 的 GitHub 代码库（https://github.com/yogthos/markdown-clj），了解该库的更多信息。

# 7.15　用Luminus来构建应用

作者：Dmitri Sotnikov

## 问题

希望很快创建一个典型的、基于 Ring/Compojure 的 Web 应用结构，快速开始新的 Web 开发项目。

## 解决方案

在创建新项目时，使用 Luminus 的 Leiningen 模板。

在命令行输入：

```
$ lein new luminus myapp
```

这将创建一个新的 Ring/Compojure 应用，带有命名空间骨架和资源目录结构，可以打包成独立的 Java 档案（JAR）文件，或可以直接部署在应用服务器上的 Web 档案（WAR）文件。

在开发模式下，启动应用只要执行：

```
$ lein ring server
```

## 讨论

虽然 Luminus 能做的事情你自己也能完成，但它提供了一组标准化的库和样板文件，用于创建常见的 Ring/Compojure 应用。

该模板在你的项目中生成了标准的目录结构，定义了应用的主处理函数，在 project.clj 文件中添加了 `lein-ring` 的钩子，提供了默认的日志配置，并且建立了默认的路由。

在创建应用时，你可以指定一些特性描述来添加功能，这些特性描述扩展了生成的代码，目的是包含相关的组件。下面是一些例子，在初始化应用时包含了不同数据库的默认配置：

```
$ lein new luminus app1 +h2

# 或用 PostgreSQL:
$ lein new luminus app2 +postgres

# 或 ClojureScript!
$ lein new luminus app3 +cljs
```

```
# 你也可以同时指定多个特性描述：
$ lein new luminus app4 +cljs +postgres
```

得到的应用在结构上使用了下面的命名空间。

`<app-name>.handler` 命名空间包含了 `init` 和 `destroy` 函数。它们分别在应用启动和关闭时被调用。它也包含了 `app` 处理函数，被 Ring 用来初始化路由处理器。

`<app-name>.routes` 命名空间用于存放应用的核心逻辑。这是你定义应用路由及其处理函数的地方。`<app-name>.routes.home` 命名空间包含了默认的 / 和 /about 页面的路由。

站点的布局是由 `<app-name>.views.layout` 命名空间中的 `render` 函数生成的。页面的 HTML 模板可以在 src/<app-name>/views/templates/ 下找到。Luminus 使用 Selmer（https://github.com/yogthos/Selmer）作为默认的模板引擎，这在 7.12 节“用 Selmer 实现模板”中介绍过。

其他各种辅助函数放在 `<app_name>.views.util` 命名空间中。

如果选择了数据库特性描述，就会创建 `<app_name>.models.db` 和 `<app_name>.models.schema` 命名空间。`schema` 命名空间为表定义而保留，`db` 命名空间存放处理应用模型的函数。

应用可以用 **`lein ring uberjar`** 打包成独立的 JAR 文件，或用 **`lein ring uberwar`** 打包成 WAR 文件。

## 参阅

- Luminus 项目主页（http://www.luminusweb.net/）。

第 8 章

# 性能与开发效率

## 8.0 简介

你已经花了一段时间来开发一个大项目：除了发布之外，接下来该做些什么？不论它是产品、内部服务，还是库，最后一步（也是最重要的一步）就是将你的劳动成果交付给目标客户。

开发者很容易忘记，代码完成只是应用的真正生命周期的开始。成功的项目将在生产阶段经历更多的时间，远超过开发阶段，稳定性与可维护性是更宝贵的特性。

本章全部内容都在探讨真正完成工作，让构建的软件能在接下来的数年里尽可能无痛苦地运行。对于任务性能、日志 、发布版本或长期维护来说，最重要的是发布真正优秀的软件。就像孩子初次脱离父母的庇护一般，我们对于新构建的软件，当然会有很多担心，我们希望这些实例能帮助你做出正确的应对。

## 8.1 AOT编译

作者：Luke VanderHart

### 问题

希望以预编译的 JVM 字节码，即 .class 文件的方式来交付你的代码，而不是以 Clojure 源代码的方式。

## 解决方案

在项目的 project.clj 文件中使用 `:aot`（事先）编译键，指明应该编译成 .class 文件的命名空间。`:aot` 键的值是一个向量，包含一些具体要编译的命名空间，或者一些正则表达式字面量，指明名称匹配的命名空间应该编译。或者不用向量，你可以用关键字 `:all` 作为值，这样将事先编译项目的所有命名空间：

```
:aot [foo.bar foo.baz]

;; or...
:aot [#"foo\.b.+"] ; 编译以 "foo.b" 开始的所有命名空间

;; or...
:aot :all
```

请注意，如果项目已经指定了 `:main` 命名空间，Leiningen 默认会事先编译它，不论它是否出现在 `:aot` 的值中。

在项目配置为 AOT 编译后，就可以在命令行执行 **`lein compile`**，对它进行编译。所有生成的类将放在 target/classes 目录下，除非用 `:target-path` 或 `:compile-path` 选项覆写了输出目录。

## 讨论

重要的是要理解，AOT 编译没有改变代码实际运行的方式。它没有变得更快，也没有变得不同。所有 Clojure 代码都会先被编译成字节码，然后再执行。AOT 编译只是说它在一个固定的时间点一次完成，而不是在程序加载和运行时按需完成。

但是，尽管没有变得更快，它在下列情况中却是非常好的工具。

- 希望交付应用的二进制代码，但不希望包含源代码。
- 希望应用的启动时间加快一点（因为 Clojure 代码就不必当场编译了）。
- 希望生成一些类，直接由 Java 加载，实现互操作。
- 针对一些平台（如 Android），它们不支持定制类加载器，在运行时执行新的字节码。

你可能会注意到，对每个 AOT 编译的命名空间，不止生成一个类文件。实际上，对每个函数，命名空间本身，以及所有附加的 `gen-class`、`deftype` 或 `defrecord` 形式，都会生成独立的 Java 类。这实际上与 Java 本身没有不同，Java 也总是将内部类编译成独立的类文件，从 JVM 的角度来看，Clojure 函数相当于匿名内部类。

## 参阅

- Clojure 关于 AOT 编译的官方文档（http://clojure.org/compilation）。
- 8.2 节“将项目打包成 JAR 文件”。

# 8.2　将项目打包成JAR文件

作者：Alan Busby

## 问题

希望将项目打包成可执行的 JAR。

## 解决方案

用 Leiningen 构建工具，将应用打包成一个 uberjar，即包含应用和所有依赖关系的 JAR 文件。

要继续本实例，先创建一个新的 Leiningen 项目：

```
$ lein new foo
```

在项目的 project.clj 文件中添加 `:main` 和 `:aot` 参数，将项目配置为可执行的：

```
(defproject foo "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]]
  :main foo.core
  :aot :all)
```

要让项目可执行，在 src/foo/core.clj 文件中添加 `-main` 函数和 `:gen-class` 声明。删除原有的 `foo` 函数：

```
(ns foo.core
  (:gen-class))

(defn -main [& args]
  (->> args
       (interpose " ")
       (apply str)
       (println "Executed with the following args: ")))
```

用 **`lein run`** 命令执行应用，验证它工作正常：

```
$ lein run 1 2 3
```

调用 **`lein uberjar`**，将应用和所有依赖关系打包：

```
$ lein uberjar
Created /tmp/foo/target/uberjar/foo-0.1.0-SNAPSHOT.jar
Created /tmp/foo/target/foo-0.1.0-SNAPSHOT-standalone.jar
```

执行生成的 target/foo-0.1.0-SNAPSHOT-standalone.jar 文件，即用 `-jar` 选项将它传递给 `java`：

```
$ java -jar target/foo-1.0.0-standalone.jar 1 2 3
Executed with the following args: 1 2 3
```

# 讨论

可执行的 JAR 文件提供了一种极好的方法将文件打包，这样就能提供给用户，由 cron 任务调用，与其他 Unix 工具组合使用，或在适用命令行的其他场景下使用。

在背后，可执行的 JAR 文件与其他 JAR 文件一样，都包含一组程序资源，如类文件、Clojure 源文件和 classpath 资源。此外，可执行的 JAR 文件还包含一些元数据，表明哪个类包含 `main` 方法，这在它内部的 manifest 文件的 `Main-Class` 标签中指明。

Leiningen 的 uberjar 是一个 JAR 文件，它不仅包含你的程序，还打包了所有的依赖关系。在 Leiningen 生成 uberjar 时，它能检测 project.clj 中的 `:main` 信息，得知程序提供了一个 `-main` 函数，生成适当的 manifest 文件，确保得到的 JAR 文件是可执行的。

命名空间中的 `:gen-class` 以及 `:aot` 的 Leiningen 选项是必要的，这样才能将 Clojure 源代码预编译成 JVM 的类，因为 manifest 文件中的“Main-Class”条目不知道如何引用或编译 Clojure 源文件。

### 打包JAR时不带依赖关系

在打包项目时，Leiningen 既可以带上依赖关系，也可以不带依赖关系。

`jar` 命令打包项目代码时，不带任何上游依赖关系。甚至 Clojure 本身也不包含在 JAR 文件内，你需要 BYOC[1]。

在 foo 项目中调用 **`lein jar`** 命令，将生成 target/foo-0.1.0-SNAPSHOT.jar：

```
$ lein jar
Created /tmp/foo/target/jar/target/foo-0.1.0-SNAPSHOT.jar
```

用 **`unzip`** 命令[2] 列出 JAR 文件的内容，你会看到打包的内容很少，只有一个 Maven 的 .pom 文件，生成的 JVM 类文件，以及项目的其他一些文件：

```
$ unzip -l target/foo-0.1.0-SNAPSHOT.jar
Archive: target/foo-0.1.0-SNAPSHOT.jar
  Length     Date   Time     Name
 --------    ----   ----     ----
      113 12-06-13 10:26   META-INF/MANIFEST.MF
     2595 12-06-13 10:26   META-INF/maven/foo/foo/pom.xml
```

注 1：带上你自己的 Clojure！

注 2：大多数类 Unix 系统都提供。

```
       91 12-06-13 10:26   META-INF/maven/foo/foo/pom.properties
      292 12-06-13 10:26   META-INF/leiningen/foo/foo/project.clj
      292 12-06-13 10:26   project.clj
      229 12-06-13 10:26   META-INF/leiningen/foo/foo/README.md
    11220 12-06-13 10:26   META-INF/leiningen/foo/foo/LICENSE
        0 12-06-13 10:26   foo/
     1210 12-06-13 10:26   foo/core$_main.class
     1304 12-06-13 10:26   foo/core$fn__16.class
     1492 12-06-13 10:26   foo/core$loading__4910__auto__.class
     1755 12-06-13 10:26   foo/core.class
     2814 12-06-13 10:26   foo/core__init.class
      162 12-04-13 14:54   foo/core.clj
 --------                  -------
    23569                  14 files
```

但是，如果列出 target/foo-0.1.0-SNAPSHOT-standalone.jar 中的文件，会看到超过 3000 个文件[3]。

因为打包的 pom.xml 文件包含了项目依赖关系的列表，所以像 Leiningen 或 Maven 这样的工具能自己解决这些依赖关系。这样就能高效地对库进行打包。你能想象每个 Clojure 库都包含它的全部依赖关系吗？那会是一场带宽噩梦。

由于这个特点，在使用 `lein deploy` 时，部署到远程储存地的就是这样精简的 JAR[4]。

因为不包含依赖关系（就是说，Clojure），所以需要多做一点工作，才能运行 foo 应用。首先，下载 Clojure 1.5.1（http://clojure.org/downloads）。然后，通过 `java` 命令调用 `foo.core`，在 classpath 中带上 clojure-1.5.1.jar 和 foo-0.1.0-SNAPSHOT.jar（通过 `-cp` 选项）。

```
# 下载 Clojure
$ wget \
  http://repo1.maven.org/maven2/org/clojure/clojure/1.5.1/clojure-1.5.1.zip
$ unzip clojure-1.5.1.zip

# 执行该应用
$ java -cp target/foo-0.1.0-SNAPSHOT.jar:clojure-1.5.1/clojure-1.5.1.jar \
       foo.core \
       1 2 3
Executed with the following args: 1 2 3
```

## 参阅

- 3.6 节“从命令行运行程序”，了解从 Leiningen 运行 Clojure 程序。
- 8.1 节“AOT 编译”。
- `lein-bin`（https://github.com/Raynes/lein-bin），一个 Leiningen 插件，生成独立的控制台可执行程序，支持 OS X、Linux 和 Windows。

---

注 3：我们不可能把所有文件都打印出来。你用 **`lein uberjar && unzip -l target/foo-0.1.0-SNAPSHOT-standalone.jar`** 命令自己看一下。

注 4：参阅 8.9 节“向 Clojars 发布一个库”，了解发布库的更多信息。

# 8.3 创建WAR文件

作者：Luke VanderHart

## 问题

希望将基于 Ring 开发的 Clojure Web 应用作为标准的 Web 档案（WAR）文件，部署到常用的 Java EE 容器中，如 Tomcat、JBoss 或 WebLogic。

## 解决方案

假定你正在用 Ring 或基于 Ring 的框架（例如 Compojure），要组织项目并构建 WAR 文件，最容易的方法就是使用 Leiningen 的 `lein-ring` 插件。假定项目有一个 Ring 处理函数，定义在命名空间 `warsample.core` 中[5]，像这样：

```
(ns warsample.core)

(defn handler [request]
  {:status 200
   :headers {"content-type" "text/html"}
   :body "<h1>Hello, world!</h1>"})
```

要用 `lein-ring` 来配置项目，请在 Leiningen 的 project.clj 文件中添加下面的键值对：

```
:plugins [[lein-ring "0.8.8"]]
:ring {:handler warsample.core/handler}
```

你也需要确保应用声明了对 `javax.servlet/servlet-api` 库的依赖关系。大多数 Web 应用库确实包含传递的依赖关系，可以通过执行 **`lein deps :tree`** 来验证这一点。如果使用的其他库没有包含它，你可以自己包含它，将 `[javax.servlet/servlet-api "2.5"]` 添加到 project.clj 文件的 `:dependencies` 键中。

`:plugins` 键指定项目使用 `lein-ring` 插件，`:ring` 键下面的映射表指定了 `lein-ring` 特定的配置选项。唯一必须的选项是 `:handler`，它指明了应用主要 Ring 处理函数的名称。

`lein-ring` 为本地运行应用提供了方便的方法，这是为了开发和测试。在命令行，只要输入：

```
$ lein ring server
```

一个嵌入式的 Jetty 服务器就会启动，为你的 Ring 应用提供服务（默认在 3000 端口，但可以在 `lein-ring` 选项中修改）。它也会打开操作系统默认的浏览器，浏览该页。

在你认为应用正确运行之后，就可以用 **`lein ring war`** 或 **`lein ring uberwar`** 命令来创建

注 5：如果你没有类似名称的项目，又希望继续本实例，就用 **`lein new warsample`** 创建一个新项目。

WAR 文件。它们都接受要生成的 WAR 文件名作为参数：

```
$ lein ring war warsample.war

$ lein ring uberwar warsample-with-deps.war
```

`lein ring war` 创建的 WAR 文件只包含你的应用代码，不包含传递的依赖关系，而 `lein ring uberwar` 创建的 WAR 文件会包含所有依赖关系的 JAR 文件。

这两个命令都会在创建 WAR 之前，生成必要的配置和连接关系（诸如 WEB-INF 目录和 Web.xml 文件）。讨论小节介绍了一些可以传递给 `lein ring` 的选项，它们将影响这些组件的生成。

在发出创建 WAR 的命令后，你会在项目的 target 目录下发现生成的 WAR 文件。这是一个相当普通的 WAR 文件，可以像标准的 J2EE WAR 文件一样部署。每个应用服务器都不一样，所以请查看你喜欢的系统的文档，看看如何部署 WAR 文件。如果你们有运营团队负责产品部署，你肯定希望与他们核对，确保遵守他们的流程和最佳实践。

# 讨论

要理解 `lein ring war` 生成的精简 WAR 文件与 `lein ring uberwar` 生成的“uberwar”之间的区别，以及何时用哪一个是至关重要的。

精简的 WAR 文件不包含项目的依赖关系，它只包含应用自身的代码。这意味着你的程序不能运行，除非你确保程序依赖的所有 JAR 文件，包括 Clojure 本身，都放在应用的共享类路径中。如何做到这一点取决于你使用的应用服务器，所以你必须参考系统的文档，确定如何提供它们。

“uberwar”则不一样，它在 WAR 档案中包含了程序依赖的所有 JAR，它们被作为绑定的库，放在 WEB-INF/lib 目录下。兼容的应用服务器能够在应用自己的类加载器上下文中运行每个应用（每个部署的 WAR 文件），并将绑定的 JAR 只提供给它们的应用。

通常，uberwar 是比较安全的选择。它让你不必手工管理库，更好地反映了应用的类路径在开发时的样子。

但 uberwar 的代价在于，一个库如果被多个应用绑定，它可能被加载多次。如果你要运行 10 个应用，假设每个都用到了 Compojure，服务器就会将 Compojure 加载到 JVM 类空间 10 次，每个应用一次。一些组织机构要求在部署时限制资源使用或提供高性能，宁愿确保应用的依赖关系中冗余最小。如果是这样，你可能不得不退而使用精简的 WAR 文件，在应用服务器的共享库中手工管理依赖关系。

## 依赖关系冲突

现代的 J2EE 应用服务器对于不同的应用，能很好地保持类路径和绑定库隔离，尽管如此，你必须注意一些场景，即应用依赖的库属于核心 J2EE 平台，例如 JDBC，Servlet API，像 JAX-*、StAX、JMS 这样的 XML 库，等等。

这些类通常是由应用服务器自己提供的，如果你的应用引用了它们，这些引用就会解析到容器提供的实例，而不是应用绑定的版本。如果它们完全一样，那很好。但如果版本不匹配，在类的 API 中引入了突破性的变更，你就可能遇到原因不明的错误，因为应用尝试调用的类与它们构建时使用的类不同。

在这种情况下，你需要调整应用容器和应用程序使用的依赖关系的版本，确保它们兼容。

### 其他`lein-ring`选项

`lein-ring` 提供了一些附加的选项，可以在 project.clj 文件的 `:ring` 配置映射表中设置，来调整 WAR 文件生成的方式。完整的描述请参考 `lein-ring` 的项目页面（https://github.com/weavejester/lein-ring）。

表 8-1 列出了一些比较有用的选项。

表8-1：`lein-ring`的WAR选项

| 键 | 描述 | 默认值 |
|---|---|---|
| `:war-exclusions` | 一系列的正则表达式，指明要排除在目标 WAR 之外的所有文件 | 所有的隐藏文件 |
| `:servlet-class` | 生成的 `Servlet` 类的名字 | |
| `:servlet-name` | web.xml 中 servlet 的名字 | 处理函数的名字 |
| `:url-pattern` | web.xml 中 servlet 映射的 URL | /* |
| `:web-xml` | 使用指定的 web.xml 文件，而不是生成的 | |

### 从头开始创建WAR文件

如果没有使用 Ring，或者有很好的理由不使用 `lein-ring` 插件，你仍然可以创建 WAR 文件，但创建过程需要更多手工工作。幸运的是，WAR 实际上就是 JAR 文件，只是扩展名不同，并且包含一些附加的内部结构和配置文件，所以你可以用标准的 `lein jar` 工具来生成，只要在档案中适当的位置添加下面的文件。

你也需要自己定义一些 AOT 类来实现 `javax.servlet.Servlet`，并让它们调用你的 Clojure 应用。然后你需要通过部署描述文件（web.xml）将它们组织起来，交给应用服务器。

WAR 文件的结构是：

```
<war root>
|-- <static resources>
```

```
|-- WEB-INF
    |-- web.xml
    |-- <app-server-specific deployment descriptors>
    |-- lib
    |   |-- <bundled JAR libraries>
    |-- classes
        |-- <AOT compiled .class files for servlets, etc.>
        |-- <.clj source files>
```

全面解释这些元素超出了本实例的范围。更多的信息请参考 Oracle 的 J2EE 指南（http://docs.oracle.com/javaee/7/tutorial/doc/packaging003.htm）中关于打包 Web 档案的部分。

其他的 Web 服务器库（例如 Pedestal Server）如果包含针对 Leiningen 的工具，通常也有创建 WAR 文件的工具，因此请查看你用的库的文档。

## 参阅

- 8.1 节“AOT 编译”。
- 8.2 节“将项目打包成 JAR 文件”。
- `lein-ring` 的项目页面（https://github.com/weavejester/lein-ring）。
- Oracle 的 J2EE 指南（http://docs.oracle.com/javaee/7/tutorial/doc/）。

# 8.4　将应用作为守护进程运行

作者：Ryan Neufeld

## 问题

希望在另一个操作系统进程中，将 Clojure 应用作为守护进程运行（即希望应用在后台运行）。

## 解决方案

用 Apache Commons Daemon 库来编写应用，可以在后台进程中执行。Daemon 库包含两个部分：`Daemon` 接口，你的应用必须实现它，以及一个系统应用[6]，它将 `Daemon` 兼容的应用作为守护进程运行。

首先在项目的 project.clj 文件中加入 Daemon 依赖关系。如果还没有项目，就用 **`lein new my-daemon`** 创建一个新项目。由于 Daemon 是基于 Java 的系统，所以先启用 AOT 编译，以便生成类文件：

注 6：Unix 系统是 `jsvc`，Windows 是 `procrun`。

```
(defproject my-daemon "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [org.apache.commons/commons-daemon "1.0.9"]]
  :main my-daemon.core
  :aot :all)
```

要实现 `org.apache.commons.daemon.Daemon` 接口，就在项目的一个命名空间中添加相应的 `:gen-class` 声明和接口函数。对于最小的函数式守护进程，要实现 `-init`、`-start` 和 `-stop`。为了得到最佳结果，提供 `-main` 函数来支持应用的冒烟测试，而不去碰 `Daemon` 接口。

```
(ns my-daemon.core
  (:import [org.apache.commons.daemon Daemon DaemonContext])
  (:gen-class
    :implements [org.apache.commons.daemon.Daemon]))

;; 应用状态的粗略近似
(def state (atom {}))

(defn init [args]
  (swap! state assoc :running true))

(defn start []
  (while (:running @state)
    (println "tick")
    (Thread/sleep 2000)))

(defn stop []
  (swap! state assoc :running false))

;; Daemon 实现

(defn -init [this ^DaemonContext context]
  (init (.getArguments context)))

(defn -start [this]
  (future (start)))

(defn -stop [this]
  (stop))

(defn -destroy [this])

;; 支持命令行调用
(defn -main [& args]
  (init args)
  (start))
```

调用 Leiningen 的 `uberjar` 命令，打包所有必要的依赖关系和生成的类：

```
$ lein uberjar
Compiling my-daemon.core
Created /tmp/my-daemon/target/my-daemon-0.1.0-SNAPSHOT.jar
Created /tmp/my-daemon/target/my-daemon-0.1.0-SNAPSHOT-standalone.jar
```

在继续进行之前，先用 java 来运行应用，对它进行测试：

```
$ java -jar target/my-daemon-0.1.0-SNAPSHOT-standalone.jar
tick
tick
tick
# 按Ctrl-C键来终止混乱状态
```

验证应用工作正常后，安装 jsvc[7]。最后也是最关键的时刻，将应用作为守护进程运行，即用所有必需的参数来调用 jsvc。参数包括 Java home 目录的绝对路径、uberjar、输出日志文件、你的 Daemon 实现所属的命名空间[8]：

```
$ sudo jsvc -java-home "$JAVA_HOME" \
            -cp "$(pwd)/target/my-daemon-0.1.0-SNAPSHOT-standalone.jar" \
            -outfile "$(pwd)/out.txt" \
            my_daemon.core
# 不存在！

$ sudo tail -f out.txt
tick
tick
tick
# 按 Ctrl-C 退出

# 添加 -stop 标记以终止守护进程
$ sudo jsvc -java-home "$JAVA_HOME" \
            -cp "$(pwd)/target/my-daemon-0.1.0-SNAPSHOT-standalone.jar" \
            -stop \
            my_daemon.core
```

如果一切正常，out.txt 将包含一些"tick"。守护进程的设置有一点难，但是一旦运行，它就工作得很好。如果在用 jsvc 启动守护进程时遇到问题，请使用 -debug 标记，输出更详细的诊断信息。

你可以在 https://github.com/clojure-cookbook/my-daemon 找到 my-daemon 项目完整的、能工作的副本。

注 7：在 OS X 上我们建议用 Homebrew（http://brew.sh/）来 brew install jsvc。如果使用 Linux，你很可能会在喜欢的包管理器中找到 jsvc 包。Windows 用户需要安装并使用 procrun（http://commons.apache.org/proper/commons-daemon/procrun.html）。

注 8：不要着急，我们马上会在 shell 脚本中记下这些。

## 讨论

不要抱有幻想，让基于 Java 的服务作为守护进程是困难的。10 多年来，Java 开发者一直在用 Apache Commons Daemon 来做这件事。为什么要用独立的 Clojure 工具来重新发明轮子呢？ Clojure 的一项核心力量就是能够为老曲调注入新生命，Daemon 就是这样的“老曲调”。

但是，并非所有的曲调都生来平等。虽然有些 Java 库只需要很少的 Java 互操作，但 Daemon 要求很多。利用 Apache Commons Daemon 让应用作为守护进程，需要做对两件事。第一件事是创建一个类来实现 `Daemon` 接口，并将它打包成 JAR 文件。`Daemon` 接口包括 4 个方法，分别在守护进程应用生命周期的不同时刻被调用。

`init(DaemonContext context)`
: 在应用初始化时被调用。这是设置应用初始状态的地方。

`start()`
: 在 `init` 之后被调用。这是开始执行工作的地方。`jsvc` 希望 `start()` 快速完成，所以你应该在一个 `future` 或 Java `Thread` 中启动工作。

`stop()`
: 在守护进程被要求停止时被调用。在这里应该停止 `start` 中启动的所有处理工作。

`destroy()`
: 在 `stop` 后被调用，但在 JVM 进程退出之前。在传统的 Java 程序中，你会在这里释放所有获取的资源。如果你已经正确地组织了应用的结构，也许可以在 Clojure 应用中跳过这一步。通过包含一个空函数来防止 `jsvc` 报怨也没有什么关系。

很容易创建一个记录（用 `defrecord`）来实现 `Daemon` 接口，但这还不够。`Jsvc` 希望实现 `Daemon` 的类放在类路径中。要做到这一点，你必须做两件事：首先，你需要让项目采用事先（AOT）编译，在 project.clj 文件中设置 `:aot :all` 就可以了。其次，你需要征用一个命名空间来产生一个类，通过 `:gen-class` 命名空间指令。具体来说，你需要生成一个类，实现 `Daemon` 接口。利用 `:gen-class` 和 `:implements` 指令，这很容易实现：

```
(ns my-daemon.core
  ;; ...
  (:gen-class
    :implements [org.apache.commons.daemon.Daemon]))
```

设置好 `my-daemon.core` 来编译生成 `Daemon` 实现类，剩下的事情就是实现方法本身了。在函数前加上连接号（如 `-start`）就是告诉 Clojure 编译器，这个函数实际上是一个 Java 方法。而且，由于 `Daemon` 的方法是实例方法，每个函数都包含一个附加的参数，代表 `Daemon` 实例。这个参数传统上表示为 `this`。

在我们简单的例子 my-daemon 中，大部分方法实现都相当简单，除了 this 外没有别的参数，将工作代理给普通的 Clojure 函数。但 -init 值得注意一下：

```
(defn -init [this ^DaemonContext context]
  (init (.getArguments context)))
```

-init 方法接受一个附加的参数：一个 DaemonContext 实例。这个参数在它的 .getArguments 属性中记录了启动守护进程的命令行参数。在实现时，-init 对 context 调用了 .getArguments 方法，将它的返回值传递给常规的 Clojure 函数 init。

为什么将每个 Daemon 实现代理给独立的 Clojure 函数呢？通过分离 Daemon 接口实现和应用的内部工作，你保留了用其他方式调用它的能力。通过这种关注点分离，测试应用就要容易得多，要么通过集成测试，要么直接调用。-main 函数利用了这些 Clojure 函数，让你能够在守护进程之外，验证应用行为的正确性。

做好了 Daemon 兼容应用的基础工作，剩下的一步就是对应用打包。Leiningen 的 uberjar 命令完成了所有必要的准备，让应用能作为守护进程运行：将 my-daemon.core 编译成类，收集依赖关系，将它们打包成独立的 JAR 文件。

最后一项要点是，你需要运行它。由于 JVM 进程通常在底层系统调用方面做得不是很好，所以 Daemon 提供了系统调用应用，即 jsvc 和 procrun，作为 JVM 和计算机操作系统的中间层。这些应用通常用 C 编写，能够执行相应的系统调用，创建一个后台进程来执行你的应用。简单起见，我们在本节的剩余部分只讨论 jsvc 工具。

这两个工具都有令人迷惑的大量配置选项，但只有几个选项是启动工作时真正需要的。至少，你必须提供独立的 JAR 的位置（-cp），Java 安装目录（-java-home），以及希望执行的类（最后一个参数）。其他相关的属性包括 -pidfile、-outfile 和 -errfile，它们分别指定了进程的 ID、STDOUT 和 STDERR 将写往何处。在类名之后的所有参数都会传递给 -init，作为 DaemonContext。

更完整的例子是：

```
$ sudo jsvc -java-home "$JAVA_HOME" \
            -cp "$(pwd)/target/my-daemon-0.1.0-SNAPSHOT-standalone.jar" \
            -pidfile /var/run/my-daemon.pid \
            -outfile "/var/log/my-daemon.out" \
            -errfile "/var/log/my-daemon.err" \
            my_daemon.core \
            "arguments" "to" "my-daemon.core"
```

在用 jsvc 启动一个守护进程后，可以通过重新运行 jsvc 并带上 -stop 选项来停止它。

由于 jsvc 在全新的进程中重新启动了你的应用，所以它没有带上原来的执行上下文。这意味着没有环境变量，没有当前工作目录，什么都没有。该进程甚至都不是由同一个用户运行的。因此，很重要的是指定 jsvc 参数时带上绝对路径，并且有正确的权限。

例如，我们选择使用 sudo 来减少麻烦，但在生产环境中，你应该建立一个独立的用户，给它限制更多的权限。执行的用户应该有权限写入 .pid、.out 和 .err 文件，并能读取 Java 和类路径。

jsvc 和类似的工具可能是变幻无常的怪兽，因为一点点错误配置都可能导致守护进程安静地失败，没有任何警告。我们强烈建议在开发和配置守护进程时使用 -debug 和 -nodetach 选项，直到你确信一切工作正常。

在你确定合适的配置后，最后的步骤就是写一个守护进程脚本，实现守护进程的管理自动化。好的守护进程脚本记录了配置参数、文件路径和常用操作，并为它们提供一个干净的、没有噪音的接口。你可以简单调用 **my-daemon start** 或 **my-daemon stop**，而不用以前执行的 jsvc 长命令。实际上，许多 Linux 发行版都使用类似的脚本来管理系统守护进程。要实现你自己的 jsvc 守护进程脚本，我们建议阅读 Sheldon Neilson 的文章“Creating a Java Daemon (System Service) for Debian using Apache Commons Jsvc”（http://www.neilson.co.za/creating-a-java-daemon-system-service-for-debian-using-apache-commons-jsvc/）。

## 参阅

- Daemon 的文档（http://commons.apache.org/proper/commons-daemon/apidocs/index.html）。
- jsvc 帮助手册的内容，通过 **jsvc -help** 命令查看。
- procrun（http://commons.apache.org/proper/commons-daemon/procrun.html），Windows 下运行 Daemon 的工具。
- lein-daemon（https://github.com/arohner/lein-daemon），Leiningen 的插件，用于创建守护进程，可以在项目中通过 lein daemon 命令来管理。
- 8.1 节“AOT 编译”，了解关于 AOT 编译的更多信息。
- 8.2 节“将项目打包成 JAR 文件”，了解如何将应用打包成可执行的 JAR。
- Meikel Brandmeyer 的博客文章“gen-class—how it works and how to use it”（https://kotka.de/blog/2010/02/gen-class_how_it_works_and_how_to_use_it.html）。
- Stuart Sierra 的 Component 库（https://github.com/stuartsierra/component），它是一个小框架，管理软件组件的生命周期。

# 8.5 利用类型暗示减轻性能问题

作者：Ryan Neufeld

## 问题

有一些函数经常被调用，你希望优化这些方法的性能。

## 解决方案

对于给定函数，增加性能最容易的方法就是消除 Java 反射。打开 `warn-on-reflection` 设置来诊断过多的反射：

```
(defn column-idx
  "Return the index number of a column in a CSV header row"
  [header-cols col]
  (.indexOf (vec header-cols) col))

(def headers (clojure.string/split "A,B,C" #","))
(column-idx headers "B")
;; -> 1

(set! *warn-on-reflection* true)

(defn column-idx
  "Return the index number of a column in a CSV header row"
  [header-cols col]
  (.indexOf (vec header-cols) col))
;; Reflection warning, NO_SOURCE_PATH:1:1 - call to indexOf can't be resolved.

;; 100000 次无暗示的执行 ...
(time (dotimes [_ 100000] (column-idx headers "B")))
;; "Elapsed time: 329.258 msecs"
```

在确定了反射之后，为参数列表添加类型暗示，将每个参数表示为 `<^Type> <arg>`：

```
(defn column-idx
  "Return the index number of a column in a CSV header row"
  [^java.util.List header-cols col]
  (.indexOf header-cols col))

;; 100000 次正确暗示的执行
(time (dotimes [_ 100000] (column-idx headers "B")))
;; "Elapsed time: 27.779 msecs"
```

如果有一组函数相互调用，虽然你正确地暗示了参数，但还是可能看到反射警告。对参数列表本身也添加类型暗示，暗示函数返回值的类型：

```
;; 作为一个简单的例子，假定你想比较两个函数调用的结果

(defn some-calculation [x] 42)

(defn same-calc? [x y] (.equals (some-calculation x)
                                (some-calculation y)))
;; Reflection warning, NO_SOURCE_PATH:1:24 - call to equals can't be resolved.
```

```
;; 现在对 some-calculation 的返回值提供类型暗示
(defn some-calculation ^Integer [x] 42)

(defn same-calc? [x y] (.equals (some-calculation x)
                                (some-calculation y)))

;; 看啊,没有反射警告了!
```

## 讨论

在高性能代码中，通常你会选择回退到 Java 来增加性能。但是，Clojure 与 Java 之间存在着阻抗不匹配。Java 是强类型的，而 Clojure 不是。因为这一点，（几乎）每次在 Clojure 中调用 Java 函数时，都需要对提供的参数类型进行反射，以便选择合适的 Java 方法来调用。对于很少调用的方法，这不是什么大事。但对于经常执行的方法，反射的开销可能迅速积累。

类型暗示跳过了这种反射。如果你暗示了一个 Java 函数的所有参数，Clojure 编译器就不再执行反射。相反，函数调用时会直接调用相应的 Java 函数。当然，如果把类型搞错了，方法就不能正确工作，错误暗示的函数会抛出类型转换异常。

如果你有一系列的值，都是统一的类型呢？对于这些情况，Clojure 提供了几种特殊的暗示，即 `^ints`、`^floats`、`^longs` 和 `^doubles`。暗示这些类型允许你传入整个数组作为 Java 函数的参数，而不用对序列执行反射。

### 未检查的数学运算

你可能注意到，Clojure 对所有的数字类型也加了额外检查，目的是避免溢出。当然，这不是没有代价，因为 Clojure 需要检查每次操作，确保没有溢出。如果你需要最高性能，碰巧又什么都不怕，那么也许可以试试未检查的数学运算[1]。

将 `unchecked-math` 设置为 `true`，禁用这种安全性，导致加、减、乘、除和 `inc/dec` 不再检查溢出。这实际上将数值运算回退到了类似 C 的状态，正整数溢出可能得到一个负数：

```
;; 有了检查的数学运算，不可能使整数溢出
(inc Long/MAX_VALUE)
;; ArithmeticException integer overflow ...

(set! *unchecked-math* true)

;; 现在整数可以自由溢出了
(inc Long/MAX_VALUE)
;; -> -9223372036854775808
```

注 1：要准确测量 `unchecked-math` 带来的性能改进，我们建议使用 Criterium（https://github.com/hugoduncan/criterium）这样的工具。用 `time` 的基准测试代码可能很难处理，常常得到误导的结果（或根本没有结果）。

但是，unchecked-math 不是绝对的，有可能装箱的类型（boxed type）溜入到运算中，导致出现检查过的数学运算。结合 unchecked-math 和类型暗示，确保数据运算真正是未检查的。当然，你自己要承担风险！

## 参阅

- 第 1 章“原生数据”。
- 8.6 节“用原生 Java 数组进行快速数学运算”。

# 8.6 用原生Java数组进行快速数学运算

作者：Jason Wolfe

## 问题

希望对大量的数据执行快速的数学运算。

## 解决方案

原生 Java 数组是紧密存放大量数字的规范方式，并能对它们进行快速运算（常常比 Clojure 序列快 100 倍）。hiphip（数组）库（https://github.com/Prismatic/hiphip）提供了快速而容易的方式，来操作 double、long、float 或 int 成员的原生数组。

开始之前，先在项目依赖关系中添加 [prismatic/hiphip "0.1.0"]，或用 lein-try 启动 REPL：

```
$ lein try prismatic/hiphip
```

使用 hiphip 的一个 amap 宏，对有类型的数组执行快速的数学运算。amap 使用了类似 doseq 的并行绑定语法：

```
(require 'hiphip.double)

(defn map-sqrt [xs]
  (hiphip.double/amap [x xs] (Math/sqrt x)))

(seq (map-sqrt (double-array (range 1000))))
;; -> (2.0 3.0 4.0)

(defn pointwise-product
  "Produce a new double array with the product of corresponding elements of
  xs and ys"
  [xs ys]
  (hiphip.double/amap [x xs y ys] (* x y)))
(seq (pointwise-product (double-array [1.0 2.0 3.0])
```

```
                         (double-array [2.0 3.0 4.0])))
;; -> (2.0 6.0 12.0)
```

要适当修改一个数组，就用 `hiphip` 的一个 `afill!` 宏：

```
(defn add-in-place!
  "Modify xs, incrementing each element by the corresponding element of ys"
  [xs ys]
  (hiphip.double/afill! [x xs y ys] (+ x y)))

(let [xs (double-array [1.0 2.0 3.0])]
  (add-in-place! xs (double-array [0.0 1.0 2.0]))
  (seq xs))
;; -> (1.0 3.0 5.0)
```

要进行较快的像 `reduce` 那样的操作，就用 `hiphip` 的一个 `areduce` 和 `asum` 宏：

```
(defn dot-product [ws xs]
  (hiphip.double/asum [x xs w ws] (* x w)))

(dot-product (double-array [1.0 2.0 3.0])
             (double-array [2.0 3.0 4.0]))
;; -> 20.0
```

我们很愿意抛出一个快速的 `time` 基准测试，来说明获得的性能提升，但对于优化来说，JVM 是一个变幻无常的怪兽。我们建议用 Criterium（https://github.com/hugoduncan/criterium）来进行基准测试，避免常见的错误。

要了解 Criterium 对 `hiphip` 的基准测试，请参见 w01fe 的 bench.clj 的要点（https://gist.github.com/w01fe/7132440）。

# 讨论

大多数时候，Clojure 的序列抽象足以确保你完成所有工作。前面的 `dot-product` 能够用普通的 Clojure 简洁地写出来，一般情况下应该优先尝试：

```
(defn dot-product [ws xs]
  (reduce + (map * ws xs))
```

但是，一旦确定瓶颈是数学运算，原生数组可能是唯一的路。前面的 `dot-product` 实现使用 `asum` 后可以快 100 多倍，这主要是因为 `map` 产生了装箱的 Java `Double` 对象的序列。除了构造中间序列的成本外，对装箱数字的所有数学运算都要比相应的原生数字慢很多。

`hiphip` 的 `amap`、`afill!`、`reduce` 和 `asum` 宏可用于 `int`、`long`、`float` 和 `double` 类型。比方说，如果希望对浮点数组应用 `reduce`，就要用 `hiphip.float/reduce`。这些宏对每种类型确定了合适的类型暗示和优化。

Clojure 也带有内建的函数（http://clojure.org/java_interop#Java%20Interop-Arrays）来操作数组，但需要更加谨慎，才能确保得到最佳性能（通过合适的类型暗示，并使用 `*unchecked-math*`）：

```
(set! *unchecked-math* true)
(defn map-inc [^doubles xs]
  (amap xs i ret (aset ret i (inc (aget xs i)))))
```

在 Clojure 使用原生数组不是为了心中的信仰：如果不做对所有事情，就很容易得到又丑、又不比简单的序列版本快（甚至更慢）的代码。要注意的最大问题是反射，它很容易把快 100 倍变成慢 10 倍，只要一个小小的打字错误或遗漏一个类型暗示。

如果你决定迎接挑战，应该始终注意以下几点：

- 虔诚地使用 `*warn-on-reflection*`，但要注意它对代码变慢的许多其他原因不会警告。
- 可靠的剖析器，或者至少是全面的基准测试套件，这是必须的。否则，你不会知道哪个函数用了 99% 的执行时间。
- 特别是如果你没用 `hiphip`，请尝试用 `*unchecked-math*`，它几乎总是使代码更快，只要你愿意放弃溢出检查的安全性。
- 如果希望数组代码在 Leiningen 下面跑得快（https://github.com/technomancy/leiningen/wiki/Faster#tiered-compilation），你可以将下面的内容添加到 project.clj 中：`:jvm-opts ^:replace []`.

## 参阅

- `hiphip` 带有全面的基准测试套件（https://github.com/Prismatic/hiphip/blob/da72a0bcfffd2b34f02ce0fb9fbefc2de1fc5d22/test/hiphip/type_impl_test.clj），针对主要原生类型，测试它和 Clojure 的数组操作，包括与手工编码的 Java 替代程序的性能比较。
- Vertigo（https://github.com/ztellman/vertigo）超越了简单的原生类型数组，采用了完全 C 风格的结构，如果你希望以最佳的性能操作结构化的数据，这可能是一个好选择（例如，不仅是 `double` 的序列）。
- 8.5 节“利用类型暗示减轻性能问题”，了解更多关于类型暗示和未检查的数学运算的内容。
- 8.7 节“用 Timbre 进行简单剖析”，了解使用 Timbre 输出代码的剖析统计数据。

# 8.7 用Timbre进行简单剖析

作者：Ambrose Bonnaire-Sergeant

## 问题

希望得到关于代码运行时间和调用次数的细粒度统计数据。

## 解决方案

用 Timbre（https://github.com/ptaoussanis/timbre）在代码中插入剖析宏，并且不会导致产品环境中的性能下降。

开始之前，先在项目依赖关系中添加 `[com.taoensso/timbre "2.6.3"]`，或用 `lein-try` 启动 REPL：

```
$ lein try com.taoensso/timbre
```

在开发时，利用 `taoensso.timbre.profiling` 命名空间中的宏来收集基准测试指标：

```
(require '[taoensso.timbre.profiling :as p])

(defn bench-me [f]
  (p/p :bench/bench-me
    (let [_ (p/p :bench/sleep
              (Thread/sleep 10))
          n (p/p :bench/call-f-once
              (f))
          _ (p/p :bench/call-f-10-times-outer
              (dotimes [_ 10]
                (p/p :bench/call-f-10-times-inner
                  (f))))]
      (iterate f n))))

(p/profile :info :Bench-f
  (bench-me
    (fn ([] (p/p :bench/no-arg-f) 100)
        ([a] (p/p :bench/one-arg-f) +))))
```

这里我们定义了一个 Clojure 函数 `bench-me`，它调用时带上一个高阶函数 `f` 作为参数，`f` 接受 0 个或 1 个参数。

Timbre 以方便的表格形式输出丰富的剖析信息：

```
2013-Aug-25 ...  Profiling :taoensso.timbre.profiling/Bench-f
                        Name  Calls    Min    Max  MAD   Mean  Time%   Time
             :bench/bench-me      1   13ms   13ms  0ns   13ms     95   13ms
                :bench/sleep      1   11ms   11ms  0ns   11ms     76   11ms
:bench/call-f-10-times-outer      1  970μs  970μs  0ns  970μs      7  970μs
          :bench/call-f-once      1  610μs  610μs  0ns  610μs      4  610μs
:bench/call-f-10-times-inner     10   20μs  214μs 35μs   39μs      3  394μs
             :bench/no-arg-f     11    5μs  163μs 26μs   20μs      2  215μs
                [Clock] Time                                     100   14ms
              Accounted Time                                     186   26ms
```

## 讨论

用 Timbre 进行剖析对纯 Clojure 来说是极好的解决方案。标准的 JVM 剖析工具，如 YourKit 和 JVisualVM，提供了关于 Java 方法的更全面信息，但带来的性能开销也更大。

Timbre 对于剖析特定区域的代码最有用，而不是将剖析作为探索工具来优化性能。由于剖析标记只是宏，所以很灵活。例如，你可以记录特定的 `if` 分支执行了多少次，完全不需要离开 Clojure，也不必通过 YourKit 或 JVisualVM 来折腾打乱过的 Clojure 函数名字。

如果觉得剖析很有用，就应该保留在代码中，好的做法是通过命名空间别名来使用剖析宏。`p` 虽然是个方便的名字，但如果不加上明确的命名空间，很容易被本地绑定掩盖。在解决方案中，我们使用了别名 `p`，所以对 `p` 的调用就变成 `p/p`。

记住，在添加剖析语句时不要犹豫：如果不启用追踪，涉及 `taoensso.timbre.profiling/p` 的代码不会带来性能损失。这意味着你可以将追踪代码留在产品代码中。如果以后希望剖析同一段代码，或者希望剖析的注释让代码更清晰，这样做就有好处。

## 参阅

- 用 Timbre 剖析（https://github.com/ptaoussanis/timbre#profiling）。

# 8.8 用Timbre记日志

作者：Alex Miller

## 问题

希望为应用代码添加日志。

## 解决方案

用 Timbre（https://github.com/ptaoussanis/timbre）来配置日志记录，为代码添加日志信息。

开始之前，先在项目依赖关系中添加 `[com.taoensso/timbre "2.7.1"]`，或用 `lein-try` 启动 REPL：

```
$ lein try com.taoensso/timbre
```

要编写输出日志信息的函数，就用 Timbre 的 `info`、`error` 等函数：

```
(require '[taoensso.timbre :as log])

(defn div-4 [n]
  (log/info "Starting")
  (try
    (/ 4 n)
    (catch Throwable t
      (log/error t "oh no!"))
    (finally
      (log/info "Ending"))))
```

div-4 函数接受一个参数，返回 4/n。

log/info 调用会生成一条“info”级别的日志输出信息。类似地，log/error 调用将生成一条“error”级别的日志输出信息。传入一个异常对象作为第一个参数，将导致调用栈也被输出。

如果用一些值来调用 div-4，导致它成功或抛出错误，就会在 REPL 中看到类似下面的输出：

```
(div-4 2)
;; -> 2
;; *out*
;; 2013-Nov-22 10:34:11 -0500 laptop INFO [user] - Starting
;; 2013-Nov-22 10:34:11 -0500 laptop INFO [user] - Ending

(div-4 0)
;; -> 2013-Nov-22 10:34:47 -0500 laptop ERROR [user] -
;;      oh no! java.lang.ArithmeticException: Divide by zero
;; -> nil
;; *out*
;; 2013-Nov-22 10:34:21 -0500 laptop INFO [user] - Starting
;; 2013-Nov-22 10:34:21 -0500 laptop ERROR [user] -
;;   oh no! java.lang.ArithmeticException: Divide by zero
;; ... Exception stacktrace
;; 2013-Nov-22 10:34:21 -0500 laptop INFO [user] - Ending
```

# 讨论

Timbre 是开始在你的代码中加入日志的好方法。使用日志库让你在将来可以指定输出去往何处，可能不止一个目的地，或者对输出用命名空间过滤。

Timbre 将日志写到配置的任意多个“appender”（输出目的地）中。默认配置是一个目的地，写到标准输出。

例如，要添加第二个文件目的地，你可以通过启用预配置的 spit 目的地，动态修改配置：

```
;; 打开它
(log/set-config! [:appenders :spit :enabled?] true)
;; 设置日志文件位置
(log/set-config! [:shared-appender-config :spit-filename] "out.log")
```

请注意，输出文件的目录必须存在，并且用户必须能够写入该文件。一旦这个配置完成，日志消息就会被同时写入控制台和该文件。

可用的日志级别有 :trace、:debug、:info、:warn、:error 和 :fatal。默认的日志级别是 :debug，因此所有级别高于或等于 :debug 的日志都会被记录（只有 :trace 是例外）。

要在运行时改变日志级别，就改变配置：

```
(log/set-level! :warn)
```

对于 Clojure 应用中的简单日志来说，虽然 Timbre 是个优秀的库，但如果你要与许多 Java 库集成，那它可能还不够。存在各种流行的 Java 日志框架和日志包装库。如果希望利用已有的 Java 日志基础设施，你可能会发现 `tools.logging` 框架更合适。

### 参阅

- Timbre 的 Readme（https://github.com/ptaoussanis/timbre/blob/master/README.md）。
- 用 `tools.logging` 记录日志（https://github.com/clojure/tools.logging/blob/master/README.md）。

## 8.9 向Clojars发布库

作者：Ryan Neufeld，最初由 Simon Mosciatti 提交

### 问题

你用 Clojure 构建了一个库，希望发布给全世界。

### 解决方案

发布库最容易的地方就是 Clojars（https://clojars.org/），它是针对开源库的社区代码库。开始之前，先注册一个账号（https://clojars.org/register）。如果你还没有 SSH 键，那么 GitHub 上的指南“Generating SSH Keys”（https://help.github.com/articles/generating-ssh-keys）就是很好的资源。

注册账号后，你就可以发布基于 Leiningen 的项目了。如果还没有要发布的项目，就用 **lein new my-first-project-<firstname>-<lastname>** 生成一个，用你自己的名称替换 <firstname> 和 <lastname>。

现在可以用 **`lein deploy clojars`** 命令，将你的库发布到 Clojars 上：

```
$ lein deploy clojars
WARNING: please set :description in project.clj.
WARNING: please set :url in project.clj.
No credentials found for clojars (did you mean `lein deploy clojars`?)
See `lein help deploy` for how to configure credentials.
Username: # ❶
Password: # ❷
Wrote .../my-first-project-ryan-neufeld/pom.xml
Created .../my-first-project-ryan-neufeld-0.1.0-SNAPSHOT.jar
Could not find metadata my-first-project-ryan-neufeld:
    .../0.1.0-SNAPSHOT/maven-metadata.xml \
    in clojars (https://clojars.org/repo/)
```

```
Sending .../my-first-project-ryan-neufeld-0.1.0-20131113.123334-1.pom (3k)
    to https://clojars.org/repo/
Sending .../my-first-project-ryan-neufeld-0.1.0-20131113.123334-1.jar (8k)
    to https://clojars.org/repo/
Could not find metadata my-first-project-ryan-neufeld:.../maven-metadata.xml \
    in clojars (https://clojars.org/repo/)
Sending my-first-project-ryan-neufeld/.../0.1.0-SNAPSHOT/maven-metadata.xml (1k)
    to https://clojars.org/repo/
Sending my-first-project-ryan-neufeld/.../maven-metadata.xml (1k)
    to https://clojars.org/repo/
```

❶ 输入你的 Clojars 用户名，然后回车。

❷ 输入你的 Clojars 口令，然后回车。

在这个命令完成后，你的库既可以在 Web 上访问（https://clojars.org/my-first-project-ryan-neufeld），也可以作为 Leiningen 的依赖关系（`[my-first-project-ryan-neufeld "0.1.0-SNAPSHOT"]`）。

## 讨论

发布一个库不会比这更简单了，只要创建一个账户并按下“大红按钮”。Leiningen 和 Clojars 共同使它变得非常容易，像你一样的 Clojure 社区开发者可以向大家发布他们的库。

在这个例子中，你发布一个简单的、唯一命名的库，不太注意版本、发布策略，也没有足够的元数据。在真正的项目中，应该注意这些事情，成为开源世界的好公民。

最容易的改变是添加合适的元数据和一个网站。在你的 project.clj 文件中，添加准确的 `:description` 和 `:url`。如果你的项目没有网站，请考虑连接到项目的 GitHub 页面（或其他公共 SCM 的“项目页面”）。

较难的是让项目有一致的版本号。我们建议的策略是语义版本（Semantic Versioning，http://semver.org/），简称 semver。这种策略用 3 个部分来描述版本，即主版本、次版本和补丁，用句点连接。结果就像“0.1.0”或“1.4.2”。每个版本位置表明一定层次的稳定性以及跨发行版的一致性。相同主版本号的发行版应该是 API 兼容的，升级主版本号就是说“我已经从根本上改变了这个库的 API”。次版本号变化表明已经添加了一些新的、向后兼容的功能。最后，补丁号表明一些缺陷得到修正。

遵守语义版本肯定需要自觉，这样做的好处是追随你的开发者比较容易理解库的版本，相信它们的行为是符合预期的。

代码签名是部署过程中的另一项重要考虑。对发布的文件签名，让用户知道这些文件是由他们信任的人（你）创建的，包含的就是你的意图（即它们没有被篡改）。Leiningen 包含了用 GPG 对发布的文件签名的机制，并在 `lein deploy` 发布时，包含了相关的 .asc 签名文件。启用代码签名在 Leiningen 的部署库指南（https://github.com/technomancy/leiningen/blob/stable/doc/DEPLOY.md#gpg）的“GNU Privacy Guard ”(GPG) 节中介绍。

## 参阅

- Clojars 的维基页面（https://github.com/ato/clojars-web/wiki），包含了大量关于向 Clojars 发布库的信息。
- Leiningen 的部署库指南（https://github.com/technomancy/leiningen/blob/master/doc/DEPLOY.md），介绍了代码签名，以及如何向 Clojars 之外的地方部署库。
- `lein help deploy` 命令的输出。

# 8.10 使用宏来简化API弃用

作者：Michael Fogus

## 问题

希望用 Clojure 宏来弃用 API 函数，并报告现有的弃用信息。

## 解决方案

如果其他程序员完成工作时依赖你维护的库，你在进行变更时理应深思熟虑。在修复缺陷和改进库的过程中，你最终会希望改变它的公共接口。对库中面向公众的部分进行修改，这不是件小事，但假定你确信必须这样做，那么你会希望弃用一些过时的函数。术语“弃用”基本上意味着某个函数应该避免使用，代之以另外的新函数。

例如，以 Clojure 的贡献库 `core.memoize`（https://github.com/clojure/core.memoize）为例。不必深入了解 `core.memoize` 做了些什么，只要知道在某一个时刻，它的面向公众的 API 是一个名为 `memo-fifo` 的函数，看起来像这样：

```
(defn memo-fifo
  ([f] ... )
  ([f limit] ... )
  ([f limit base] ... ))
```

显然，实现已经被省略，只突出后面的版本计划修改的部分，即该函数的名称和可用的参数向量。新 API 的细节并不重要，但它们的差异足以引起用户的迷惑。在这种情况下，如果在新版本中仅仅进行修改而不提供适当的通知，就不是好方式，真的会导致痛苦。

随之而来的问题是：如果一项功能计划要弃用，既需要临时支持原有的代码，又要警告库的用户将来会有突破性变化，你该怎么做？在本节中，我们将探讨利用宏来提供一种很好的机制，以最小的代价，弃用库的功能和宏。

例如计划弃用的 `memo-fifo`，新的函数名字就叫 `fifo`，不仅需要改变名称，也要改变参数数目。在弃用库中某些部分时，最好是打印出警告信息，指出新的、更为理想的替代函

数。因此，为了弃用 memo-fifo，先创建下面的函数 !!，来打印警告信息：

```
(defn ^:private !! [c]
  (println "WARNING - Deprecated construction method for"
           c
           "cache; preferred way is:"
           (str "(clojure.core.memoize/" c
                " function <base> <:"
                c "/threshold num>)")))
```

传入一个符号时，!! 函数打印出这样的信息：

```
(!! 'fifo)

;; WARNING - Deprecated construction method for fifo cache;
;; preferred way is:
;; (clojure.core.memoize/fifo function <base> <:fifo/threshold num>)
```

弃用信息不仅指出调用的函数被弃用了，也指出了应该使用的替代函数。对弃用消息来说，这已经很实在了，虽然你的目的可能是别的东西。不管怎样，为了在每次调用 memo-fifo 时插入这条警告，我们可以创建一个简单的宏，在函数定义体中插入对 !! 的调用，像这样：

```
(defmacro defn-deprecated [nom _ alt ds & arities]
  `(defn ~nom ~ds                                   ; ❶
     ~@(for [[args body] arities]                   ; ❷
         (list args `(!! (quote ~alt)) body))))     ; ❸
```

❶ 创建 defn 调用，带上给定的名称和文档字符串。

❷ 对给定函数参数列表进行循环。

❸ 在函数体开始的位置插入对 !! 的调用。

我们会在后面的讨论环节探讨 defn-deprecated 宏的目的，但现在，你可以看到它是怎么工作的：

```
(defn-deprecated memo-fifo :as fifo
  "DEPRECATED: Please use clojure.core.memoize/fifo instead."
  ([f] ... )
  ([f limit] ... )
  ([f limit base] ... )
```

对于 memo-fifo 的定义，仅有的改变是使用 defn-deprecated 宏代替了 defn，使用了 :as fifo 指令，并增加（或改变）了文档字符串来描述弃用。defn-deprecated 宏负责组装宏体的各个部分，在使用时打印出警告：

```
(def f (memo-fifo identity 32))
;; WARNING - Deprecated construction method for fifo cache;
;; preferred way is:
;; (clojure.core.memoize/fifo function <base> <:fifo/threshold num>)
```

每次调用 memo-fifo 时，警告信息只会显示一次，由于该函数的性质，这应该够了。

## 讨论

除了用宏之外，还有一些不同的方法来处理这种情况。例如，!! 函数可以接受一个函数和一个符号作为参数，包装该函数，插入弃用的警告信息：

```
(defn depr [fun alt]
  (fn [& args]                                    ; ❶
    (println
      "WARNING - Deprecated construction method for"
      alt
      "cache; preferred way is:"
      (str "(clojure.core.memoize/" alt
           " function <base> <:"
           alt "/threshold num>)"))
    (apply fun args)))                            ; ❷
```

❶ 在调用弃用的函数之前，先返回一个函数打印弃用信息。

❷ 调用弃用的函数。

这个 !! 的新实现将以这种方式工作：

```
(def memo-fifo (depr old-memo-fifo 'fifo))
```

此后，调用 memo-fifo 函数将打印出弃用信息。像这样使用高阶函数是一种合理的方式，目的是避免使用宏的潜在复杂性。但是，出于某些原因，我们选择使用宏的版本，后续小节将解释。

#### 保持调用栈

说实话，Clojure 产生的异常调用栈有时候可能令人痛苦。如果你决定使用 depr 这样的高阶函数，就要注意，如果执行时产生异常，就会另外加一层调用栈。使用 !! 这样的宏，将操作直接代理给 defn，就可以确保调用栈不会掺杂（可以这么说）。

#### 元数据

使用像 defn-deprecated 这样近乎一对一的替换宏，让你保持函数的元数据。请看：

```
(defn-deprecated ^:private memo-foo :as bar
  "Does something."
  ([] 42))

(memo-foo)
;; WARNING - Deprecated construction method for bar cache;
;; preferred way is:
;; (clojure.core.memoize/bar function <base> <:bar/threshold num>)
;;=> 42
```

因为 defn-deprecated 将它的部分行为延迟到 defn，所以它的元素上附着的所有元数据都自动传递下去，像期望的一样：

```
(meta #'memo-foo)

;;=> {:arglists ([]), :ns #<Namespace user>,
;;    :name memo-foo, :private true, :doc "Does something.",
;;    ...}
```

使用高阶函数的方式不会自动保持元数据：

```
(def baz (depr foo 'bar))

(meta #'baz)
;;=> {:ns #<Namespace user>, :name baz, ...}
```

当然，如果有必要，你可以复制元数据，但如果宏的方式能替你做到，为什么还要这么做呢？

### 更快的调用现场

depr 函数由于需要处理你给它的任意函数，所以在内部需要使用 apply。虽然在 core.memoize 函数的例子中这不是问题，但对于要求更高性能的函数，这可能成为问题。不过在实际中，使用 println 可能超过 apply 的代价，所以如果你真需要弃用一个高性能函数，可能需要考虑下面的方法。

### 编译时警告

defn-deprecated 的操作方式，是在函数每次调用时打印出弃用信息。如果函数要求很高的速度，这可能会有问题。很少有比控制台打印更影响函数速度的东西。因此，你可以稍稍改变 defn-deprecate，让它在编译时报警，而不是在运行时。

```
(defmacro defn-deprecated [nom _ alt ds & arities]
  (!! alt)                      ; ❶
  `(defn ~nom ~ds ~@arities))  ; ❷
```

❶ 在访问宏时打印警告信息。

❷ 将函数定义代理给 defn，不作掺杂。

请看编译时的警告：

```
(defn-deprecated ^:private memo-foo :as bar
  "Does something."
([] 42))

;; WARNING - Deprecated construction method for bar cache;
;; preferred way is:
;; (clojure.core.memoize/bar function <base> <:bar/threshold num>)
;;=> #'user/memo-foo
```

```
(memo-foo)
42
```

如果你发行的库是作为源代码，而不是作为编译好的程序，这种方法将会很有效。

### 关闭它

宏真正的优美之处不是它们允许你改变程序的语义，而是它们允许你在不合适的时候，避免这样做。例如，如果使用宏，你可以在编译时运行所有 Clojure 可用的代码。非常感谢，整个 Clojure 语言在编译时都可用。因此，我们可以检查作为元数据附着在命名空间上的布尔标记，决定是否报告编译时的弃用信息。我们可以修改最新的 `defn-deprecated`，来展示这种技巧：

```
(defmacro defn-deprecated
  [nom _ alt ds & arities]
  (let [silence? (:silence-deprecations (meta clojure.core/*ns*))] ; ❶
    (when-not silence?  ; ❷
     (!! alt)))
  `(defn ~nom ~ds ~@arities))
```

❶ 查看当前命名空间的元数据。

❷ 只在标记没有被设置为 silence 模式时，报告弃用警告。

`defn-deprecated` 宏检查了当前命名空间中 `:silence-deprecations` 元数据属性的状态，根据它决定是否报告弃用警告。如果最终使用了这种方式，那么就可以对命名空间关闭弃用警告，只要在 `ns` 声明中添加下面的内容：

```
(ns ^:silence-deprecations my.awesome.lib)
```

现在，在该命名空间使用 `defn-deprecated` 就不会打印出警告。将来版本的 Clojure 会提供更清晰的方式，来创建和管理编译时的标记，但现在这是一种体面的折中。

## 参阅

- 官方宏文档（http://clojure.org/macros）。

第 9 章

# 分布式计算

## 9.0 简介

由于存储器越来越便宜，我们倾向于存储越来越多的数据。也由于数据量不断增加，充分利用它们变得越来越困难。因此，过去十年出现了无数新技术，来处理如此大量的数据。

本章主要关注其中一种技术，即 MapReduce（http://research.google.com/archive/mapreduce.html），它是 Google 在本世纪初发展起来的。这种技术从名字上看就是函数式的，它在多个机器上并行使用 `map` 和 `reduce`，规模很大，以惊人的速度处理数据。在本章中，我们将介绍 Cascalog（http://cascalog.org/），它是在 Hadoop（http://hadoop.apache.org/）基础上建立的数据处理库。Hadoop 是开源的 MapReduce 实现。

我们也会简单介绍 Storm（http://storm-project.net/），它是一个实时的流处理库，被 Twitter、Groupon 和 Yahoo! 等技术巨头采用。

### Cascalog

Cascalog 基于 Datalog 定义了一个 DSL，Datalog 是支持 Datomic（http://www.datomic.com/）的查询语言。初看起来它可能有点奇怪，但你很快就能用 Datalog 来思考。尝试过这些实例后，请访问 Cascalog 的维基页面（https://github.com/nathanmarz/cascalog/wiki），了解更多的信息，以编写自己的查询。

Cascalog 提供了简明的语法来描述数据处理工作。转换和聚合在 Cascalog 中很容易表示，连接特别简单。你可能会非常喜欢 Cascalog 的语法，甚至在本地任务中使用它。

你可以用几种不同的方式执行 Cascalog 任务。最容易的方式就是在本地执行任务。如果这样做，Cascalog 会使用 Hadoop 的本地模式，在你自己的计算机上完成整个任务。这样既享受到并行的好处，又省去了建立集群的麻烦。

一旦你的任务超出了本地模式，就需要将它们运行在 Hadoop 集群上。拥有自己的集群有很多乐趣，但建立和维护它需要大量的工作（和金钱！）。如果你不是经常用到集群，可以考虑将任务运行在 Amazon Elastic MapReduce（EMR，http://aws.amazon.com/cn/elasticmapreduce/）上。EMR 提供了按需计算的 Hadoop 集群，就像 EC2 提供按需服务器一样。你需要有 Amazon Web Services 账户来运行任务，但这并不困难。稍后在 9.7 节“在弹性 MapReduce 上运行 Cascalog 任务”中，你会看到具体怎么做。无论是在 EMR 或自己的集群上运行任务，你都要将代码打包成 uberjar（参阅 8.2 节），然后将它发送给 Hadoop 执行。让几百个计算机为你的任务而工作，简单得让人吃惊。

# 9.1 用Storm构建活动推送系统

作者：Travis Vachon

## 问题

希望构建一个活动流处理系统，以便对应用用户生成的原始事件数据进行过滤和聚合。

## 解决方案

流是向现代因特网用户展示信息的一种主要隐喻。它被 Facebook 和 Twitter 这样的站点以及 Instagram 和 Tinder 这样的移动应用所采用。它是一种优雅的工具，为用户提供了一个窗口，查看他们每天使用的应用所产生的信息洪流。

作为这些应用的开发者，你需要工具来处理用户动作所产生的原始事件数据。这些工具必须提供过滤与聚合数据的强大能力，必须能够任意伸缩，给不断增长的用户提供服务。在理想情况下，它们应该提供高层抽象，帮助你组织和发展复杂的流处理逻辑，以适应新的特征和复杂的世界。

Clojure 在 Storm（http://storm-project.net/）中提供了这样的工具。Storm 是一个分布式实时计算系统，目的是能够实时计算 Hadoop 需要分批计算的任务。在本节中，你将构建一个简单的活动流处理系统，它能够容易地扩展，以解决许多现实问题。

首先，用 Leiningen 模板创建一个新的 Storm 项目（http://storm.incubator.apache.org/）：

```
$ lein new cookbook-storm-project feeds
```

在项目目录中，运行默认的 Storm 拓扑结构（它已由 `lein` 模板生成）：

```
$ cd feeds
$ lein run -m feeds.topology/run!
Compiling feeds.TopologySubmitter
...
Emitting: spout default [:bizarro]
Processing received message source: spout:4, stream: default, id: {}, [:bizarro]
Emitting: stormy-bolt default ["I'm bizarro Stormy!"]
Processing received message source: stormy-bolt:5,
  stream: default, id: {}, [I'm bizarro Stormy!]
Emitting: feeds-bolt default ["feeds produced: I'm bizarro Stormy!"]
```

这个生成的示例拓扑结构只是不连贯地胡乱发出示例消息，这也许不是你想要的，所以开始要修改“喷嘴”（spout），以便产生真实的事件。

按照 Storm 的说法，“喷嘴”是一个组件，它将数据插入到处理系统中，创建一个数据流。打开 src/feeds/spouts.clj，用新的“喷嘴”替换掉 `defspout` 形式，它将定期产生随机的用户事件，就像人们在在线商店中看到的一样（当然，在真正的应用里，你会将它挂接到某个真正的数据源，而不是随机数据生成器）：

```
(defspout event-spout ["event"]
  [conf context collector]
  (let [events [{:action :commented, :user :travis, :listing :red-shoes}
                {:action :liked, :user :jim, :listing :red-shoes}
                {:action :liked, :user :karen, :listing :green-hat}
                {:action :liked, :user :rob, :listing :green-hat}
                {:action :commented, :user :emma, :listing :green-hat}]]
    (spout
     (nextTuple []
       (Thread/sleep 1000)
       (emit-spout! collector [(rand-nth events)])))))
```

接下来，打开 src/feeds/bolts/clj。添加一个螺栓（bolt），它接受一个用户和一个事件，为系统中的每个用户产生一个 `(user, event)` 元组。螺栓消费一个流，进行某种处理，然后产生一个新流：

```
(defbolt active-user-bolt ["user" "event"] [{event "event" :as tuple} collector]
       (doseq [user [:jim :rob :karen :kaitlyn :emma :travis]]
    (emit-bolt! collector [user event]))
  (ack! collector tuple))
```

现在添加一个螺栓，它接受一个用户和一个事件，当且仅当该用户关注了触发该事件的用户时，它才产生一个元组：

```
(defbolt follow-bolt ["user" "event"] {:prepare true}
  [conf context collector]
  (let [follows {:jim #{:rob :emma}
                 :rob #{:karen :kaitlyn :jim}
                 :karen #{:kaitlyn :emma}
                 :kaitlyn #{:jim :rob :karen :kaitlyn :emma :travis}
                 :emma #{:karen}
```

```
                 :travis #{:kaitlyn :emma :karen :rob}}]
      (bolt
       (execute [{user "user" event "event" :as tuple}]
                (when ((follows user) (:user event))
                  (emit-bolt! collector [user event]))
                (ack! collector tuple)))))
```

最后，再添加一个螺栓，它接受一个用户和一个事件，将事件保存在一个集的散列中，就像 `{:user1 #{event1 event2} :user2 #{event1 event2}}`。这是你要展示给用户的活动流：

```
(defbolt feed-bolt ["user" "event"] {:prepare true}
  [conf context collector]
  (let [feeds (atom {})]
    (bolt
     (execute [{user "user" event "event" :as tuple}]
              (swap! feeds #(update-in % [user] conj event))
              (println "Current feeds:")
              (clojure.pprint/pprint @feeds)
              (ack! collector tuple)))))
```

这给出了所有需要的组件，但你还需要将它们组装成一个计算拓扑结构。打开 src/feeds/topology.clj，利用拓扑结构 DSL，将喷嘴和螺栓连接在一起。

```
(defn storm-topology []
  (topology
   {"events" (spout-spec event-spout)}

   {"active users" (bolt-spec {"events" :shuffle} active-user-bolt :p 2)
    "follows" (bolt-spec {"active users" :shuffle} follow-bolt :p 2)
    "feeds" (bolt-spec {"follows" ["user"]} feed-bolt :p 2)}))
```

你还需要更新文件中的 `:require` 语句：

```
(:require [feeds
           [spouts :refer [event-spout]]
           [bolts :refer [active-user-bolt follow-bolt feed-bolt]]]
          [backtype.storm [clojure :refer [topology spout-spec bolt-spec]]
                          [config :refer :all]])
```

再次运行这个拓扑结构。推送的事件将由拓扑结构中最后的螺栓在控制台打印出来：

```
$ lein run -m feeds.topology/run!
```

# 讨论

Storm 的 Clojure DSL 看起来不像标准的 Clojure。相反，它利用了 Clojure 的宏，将语言扩展到流处理领域。Storm 的流处理抽象包含四个核心原语。

- 元组（tuple）

  允许程序员为值提供名称。元组是动态类型的值列表。

- 喷嘴（spout）

  产生元组，常常通过读取分布式队列来实现。

- 螺栓（bolt）

  接受元组作为输入并产生新的元组，因为它们是 Storm 拓扑结构中的核心计算单元。

- 流（stream）

  用于连接喷嘴和螺栓以及螺栓和螺栓，创建计算拓扑结构。流可以配置一些规则，决定将特定类型的元组路由到某些螺栓实例。

下面的几个小节回顾了我们系统的组件，以便更好地了解这些原语如何一起工作：

**event-spout**

defspout 看起来很像 Clojure 的标准 defn，只有一个差别，defspout 的第二个参数是一个名字列表，将分配给这个喷嘴产生的每个元组的元素。这让你能够将元组和向量或映射表互换使用。defspout 的第三个参数是一个参数列表，将被绑定到 Storm 的操作基础设施的各个组件上。

在 event-spout 喷嘴的例子中，只用到了 collector：

```
(defspout event-spout ["event"]
  [conf context collector]
```

在喷嘴实例创建时，defspout 的主体将被求值一次，这让你有机会创建内存中的状态。通常这会连接到一个数据库或一个分布式队列，但在这个例子中，你创建了一个事件列表，包含了这个喷嘴将产生的事件：

```
 (let [events [{:action :commented, :user :travis, :listing :red-shoes}
               {:action :liked, :user :jim, :listing :red-shoes}
               {:action :liked, :user :karen, :listing :green-hat}
               {:action :liked, :user :rob, :listing :green-hat}
               {:action :commented, :user :emma, :listing :green-hat}]]
```

这个对 spout 的调用创建了一个喷嘴实例，使用了给定的 nextTuple 实现。这个实例只是休眠了一秒钟，然后用 emit-spout! 产生一个元素的元组，包含一个随机的、来自前面列表中的事件：

```
(spout
 (nextTuple []
   (Thread/sleep 1000)
   (emit-spout! collector [(rand-nth events)]))))
```

nextTuple 将在一个紧凑的循环中被反复调用，所以如果你创建了一个喷嘴来轮询外部资源，可能需要提供自己的补偿算法，避免对该资源产生过重的负载。

你也可以实现喷嘴的 ack 方法，来实现“可靠的”喷嘴，它将提供消息处理的保证。关于可靠喷嘴的更多信息，参见针对 Kestrel 队列系统的 Storm 喷嘴实现，storm-kestrel（https://github.com/nathanmarz/storm-kestrel）。

**active-user-bolt**

每次用户对系统执行一个动作，系统都需要判断其他用户是否会对它有兴趣。对于像 Twitter 这样的简单兴趣系统，用户表达兴趣只有一种方式（即关注），你可以简单地查看执行动作用户的关注列表，相应地更新推送。但在更复杂的系统中，兴趣可能表示为喜欢过的物件被执行了动作，关注的集合中添加了物件，或关注了物件的卖家。在这个世界中，你需要考虑不同的因素，针对系统中的每个用户，针对每个事件，决定事件是否应该加入用户的推送。

第一个螺栓启动了这个过程，在每次 event-spout 生成一个事件时，该螺栓针对系统中的每个用户，生成一个 (user, event) 元组：

```
(defbolt active-user-bolt ["user" "event"] [{event "event" :as tuple} collector]
  (doseq [user [:jim :rob :karen :kaitlyn :emma :travis]]
    (emit-bolt! collector [user event]))
  (ack! collector tuple))
```

defbolt 的签名看起来很像 defspout。第二个参数是一个名字列表，将分配给这个螺栓产生的元组，第三个参数是一个参量列表。第一个参量将绑定到输入元组，可以像映射表或向量那样解构。

这个螺栓的主体对系统中的用户列表进行迭代，对每个用户产生一个元组。主体的最后一行对元组调用了 ack!，这让 Storm 追踪消息处理，并在合适的时候重启处理。

**follow-bolt**

下一个螺栓是一个“准备好的螺栓”，这就是说，它保持内存状态。在许多情况下，这意味着连接到一个数据库或一个队列，或者一个数据结构，能聚合它处理的元组的某些方面，但这个例子将系统中完整的关注者列表保持在内存中。

这个螺栓看起来更像喷嘴的定义。第二个参数是一个名字的列表，第三个参数是一个螺栓配置选项的映射表（重要的是，将 :prepare 设置为 true），第四个参数是一个操作参数集，和 defspout 中接收到的一样：

```
(defbolt follow-bolt ["user" "event"] {:prepare true}
  [conf context collector]
```

螺栓的主体先定义了关注者列表，然后在对 bolt 的调用中，提供了实际的螺栓定义：

```
(let [follows {:jim #{:rob :emma}
               :rob #{:karen :kaitlyn :jim}
```

```
                :karen #{:kaitlyn :emma}
                :kaitlyn #{:jim :rob :karen :kaitlyn :emma :travis}
                :emma #{:karen}
                :travis #{:kaitlyn :emma :karen :rob}}]
(bolt
 (execute [{user "user" event "event" :as tuple}]
          (when ((follows user) (:user event))
            (emit-bolt! collector [user event]))
          (ack! collector tuple)))))
```

请注意，在这里，元组参数是在螺栓的 `execute` 定义之内，可以像平常一样解构。如果事件的用户没有关注这个元组中的用户，那么它就不会产生新的元组，只是告知它收到了输入消息。

前面曾提到，这个特殊系统的实现可以更简单，只需查询某个记录了关注关系的数据库，并在每个关注者的推送中添加一个故事。但是，因为预期会有更复杂的系统，所以提供了一个可以大规模扩展的架构。这个螺栓可以很容易扩展为一组评分螺栓，每个都基于自己的判据对 user/event 进行评估，并产生 (`user，event，score`) 元组。评分聚合螺栓将接收每个评分螺栓的评分，在收到系统中各种类型的评分螺栓的评分后，选择产生一个元组。在这个世界中，调整决定用户推送构成的因素以及它们的相对权重就很容易。实际上，以作者的观点来看，这种系统的生产体验是令人高兴的（参见 GitHub 的 Rising Tide 项目页面 https://github.com/utahstreetlabs/risingtide）。

**feed-bolt**

最后的螺栓聚合事件，形成推送。由于它只接受“评分系统”批准的 (`user， event`) 元组，所以只需要将事件添加到原来为指定用户接收的事件列表。

```
(let [feeds (atom {})]
  (bolt
   (execute [{user "user" event "event" :as tuple}]
            (swap! feeds #(update-in % [user] conj event))
            (println "Current feeds:")
            (clojure.pprint/pprint @feeds)
            (ack! collector tuple))))
```

在收到新事件时，这个玩具似的拓扑结构只是将当前的推送打印出来，但在真实世界中，可以将推送持久到一个数据存储库或缓冲中，以便高效地将推送发送给用户。

请注意，这个设计很容易扩展到支持事件融合，它不是将每个事件分开存储，而是可以聚合进入的事件和其他类似事件，为用户提供方便。

正如前面描述的，这个系统有一个大缺陷：默认情况下，Storm 元组被传送到每种螺栓的一个实例，而螺栓的实例数不是在螺栓实现中定义的。如果这个拓扑结构的操作者添加了多个 `feed-bolt`，可能会让同一个用户的事件传送到不同的螺栓实例，给每个螺栓提供同一个用户的不同推送。

让人高兴的是，这个缺陷因 Storm 支持*流分组*（stream grouping）而得到解决。流分组在 Storm 拓扑结构定义中指定。

### 拓扑结构

拓扑结构定义是最具挑战性的地方。喷嘴连接到螺栓，螺栓连接到其他螺栓，可以对它们之间的元组流进行配置，提供对计算有意义的属性。

这也是定义拓扑结构中组件级并行的地方，它为系统真正的操作并行提供了一张粗略的草图。

拓扑结构定义包括喷嘴规格说明和螺栓规格说明，它们都是一个名字到规格说明的映射表。

喷嘴规格说明只是为喷嘴实现提供了一个名字：

```
{"events" (spout-spec event-spout)}
```

可以配置多个喷嘴，规格说明可以定义喷嘴的并行：

```
{
  "events" (spout-spec event-spout)
  "parallel-spout" (spout-spec a-different-more-parallel-spout :p 2)
}
```

这个定义意味着该拓扑结构具有一个 `event-spout` 实例和两个 `a-different-more-parallel-spout` 实例。

螺栓定义有一点复杂：

```
"active users" (bolt-spec {"events" :shuffle} active-user-bolt :p 2)
"follows" (bolt-spec {"active users" :shuffle} follow-bolt :p 2)
```

像喷嘴规格说明一样，必须为螺栓提供一个名字，并指定它的并行。此外，螺栓需要指定*流分组*，它定义了：(a) 该螺栓从哪个组件接收元组，(b) 系统如何选择内存中哪个螺栓实例，向它发送元组。这两个例子中都指定了 `:shuffle`，它的意思是来自“events”的元组将被随机地送到一个 `active-user-bolt` 实例，来自“active users”的元组将被随机地送到一个 `follow-bolt` 实例。

前面曾提到，`feed-bolt` 需要更谨慎：

```
"feeds" (bolt-spec {"follows" ["user"]} feed-bolt :p 2)
```

这个螺栓规格说明指定针对“user”进行*字段分组*（fields grouping）。这意味着具有相同“user”值的所有元组都会发送到同一个 `feed-bolt` 实例。这个流分组配置了一个字段名称列表，所以字段分组考虑多个字段值的相等性，然后决定哪个螺栓实例来处理给定的元组。

Storm 的流分组也支持将元组发送到所有实例和分组，或者让螺栓产生一个元组来决定将它发送到哪里。结合已经看到的分组方式，它们提供了极大的灵活性，来决定数据如何流过你的拓扑结构。

每个组件规格说明都支持并行选项。因为拓扑结构没有指定它运行的物理硬件，这些暗示不能用于确定系统真正的并行，但集群可以利用它们来确定创建多少指定组件的内存实例。

### 部署

Storm 真正的魔力来自于部署。Storm 提供了工具，让你构建小的、独立的组件，且并未假定在同一个拓扑结构中有多少个同样的实例在运行。这意味着该拓扑结构本身的伸缩性实际上是无限的。系统边界与队列或数据库这样的外部组件之间实现数据收发，它们不需要可伸缩，但在许多情况下，大家都很清楚如何让这些服务可伸缩。

Storm 库内建一个简单的部署策略：

```
(doto (LocalCluster.)
  (.submitTopology "my first topology"
                   {TOPOLOGY-DEBUG (Boolean/parseBoolean debug)
                    TOPOLOGY-WORKERS (Integer/parseInt workers)}
                   (storm-topology)))
```

`LocalCluster` 是 Storm 集群的内存实现。你可以指定工作者（worker）的数目，它利用这些工作者来执行拓扑结构的组件，并提交该拓扑结构本身，从此它开始轮询该拓扑结构的喷嘴的 `nextTuple` 方法。随着喷嘴产生元组，它们被传送通过系统，完成该拓扑结构的计算。

向一个配置好的集群提交拓扑结构差不多同样简单，就像你在 src/feeds/TopologySubmitter.clj 中看到的：

```
(defn -main [& {debug "debug" workers "workers" :or {debug "false" workers "4"}}]
  (StormSubmitter/submitTopology
   "feeds topology"
   {TOPOLOGY-DEBUG (Boolean/parseBoolean debug)
    TOPOLOGY-WORKERS (Integer/parseInt workers)}
   (storm-topology)))
```

这个文件利用了 Clojure 的 Java 互操作性来产生一个有 `main` 方法的 Java 类。因为 project.clj 文件指定这个文件应该事先编译，所以在用 `lein uberjar` 来生成适合提交给集群的 JAR 时，这个文件会被编译，看起来就像普通的 Java 类文件。你可以将这个 JAR 上传到运行 Storm 的 Nimbus 守护进程的机器，用 `storm` 命令来提交执行：

```
$ storm jar path/to/thejariuploaded.jar feeds.TopologySubmitter "workers" 5
```

这个命令告诉集群分配 5 个专门的工作者给这个拓扑结构，开始对它的所有喷嘴轮询

`nextTuple`，就像你在使用 `LocalCluster` 时它做的一样。集群可以同时运行任意数目的拓扑结构，因为每个工作者是一个物理 JVM，最终可能运行许多不同螺栓和喷嘴的实例。

设置和运行 Storm 集群的全部细节超出了本实例的范围，但它们在 Storm 的维基页面上有大量的文档。

**结论**

我们只接触到 Storm 提供的一部分功能。它内建的分布式远程过程调用，允许用户利用 Storm 集群的力量发出同步请求，触发几百台或几千台机器的一阵活动。它保证数据得到处理的语义，允许用户构建极为健壮的系统。三叉戟（Trident）这个基于 Storm 原语的高层抽象，为复杂的实时计算问题提供了简单得惊人的解决方案。它有详细的运行时控制台，针对完全可运营的 Storm 集群，为运行时特征提供了关键的信息。这个系统提供的能力确实不同寻常。

Storm 也是一个极好的例子，展示了 Clojure 扩展到问题域的能力。它的构造扩展了 Clojure 的语法，并且符合语言习惯，让程序员停留在实时处理的领域内，不需要面对低层语言的繁文缛节。这使得 Storm 真正退到幕后。在书写良好的 Storm 拓扑结构的代码集中，大多数代码专注于手头的问题。结果是简洁的、可维护的代码以及快乐的程序员。

## 参阅

- Storm 的网站（http://storm-project.net/）。
- Storm 项目模板（https://github.com/travis/lein-storm-project-template）。
- `storm-deploy`（https://github.com/nathanmarz/storm-deploy），一个 Storm 简单部署的工具。
- Rising Tide（https://github.com/utahstreetlabs/risingtide），本实例基于的推送生成服务。

# 9.2 用抽取转换加载（ETL）管道来处理数据

作者：Alex Robbins

## 问题

需要将大量数据的格式从 JSON 列表改为 CSV，以便进一步处理。例如，需要将这样的输入：

```
{"name": "Clojure Programming", "authors": ["Chas Emerick",
                                            "Brian Carper",
                                            "Christophe Grand"]}
{"name": "The Joy of Clojure", "authors": ["Michael Fogus", "Chris Houser"]}
```

转成这样的输出：

```
Chas Emerick,Brian Carper,Christophe Grand
Michael Fogus,Chris Houser
```

## 解决方案

Cascalog 允许你编写分布式处理任务，小的任务可以在本地运行，大的任务可以在 Hadoop 集群上运行。

要继续本实例，先创建一个新的 Leiningen 项目：

```
$ lein new cookbook
```

修改新项目的 project.clj 文件，添加 `cascalog` 的依赖关系，设置 `:dev` 特性描述，对 `cookbook.etl` 命名空间启用 AOT 编译。你的 project.clj 文件现在看起来应该像这样：

```
(defproject cookbook "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [cascalog "1.10.2"]
                 [org.clojure/data.json "0.2.2"]]
  :profiles {:dev {:dependencies [[org.apache.hadoop/hadoop-core "1.1.2"]]}}
  :aot [cookbook.etl])
```

创建 src/cookbook/etl.clj 文件，并向它添加查询：

```
(ns cookbook.etl
  (:require [cascalog.api :refer :all]
            [clojure.data.json :as json]))

(defn get-vec
  "Wrap the result in a vector for Cascalog to consume."
  [m k]
  (vector
   (get m k)))

(defn vec->csv
  "Turn a vector into a CSV string. (Not production quality)."
  [v]
  (apply str (interpose "," v)))

(defmain Main [in out & args]
  (?<-
    (hfs-textline out :sinkmode :replace)
    [?out-csv]
    ((hfs-textline in) ?in-json)
    (json/read-str ?in-json :> ?book-map)
    (get-vec ?book-map "authors" :> ?authors)
    (vec->csv ?authors :> ?out-csv)))
```

创建输入数据文件 samples/books/books.json：

```
{"name": "Clojure Cookbook", "authors": ["Ryan", "Luke"]}
```

这个解决方案的全部内容都在 GitHub 的 Cascalog samples 代码库（https://github.com/clojure-cookbook/cascalog-samples）中。

要获取这个工作项目的副本，就从 GitHub 复制该项目，并签出 `etl-sample` 分支。

```
$ git clone https://github.com/clojure-cookbook/cascalog-samples.git
$ cd cascalog-samples
$ git checkout etl-sample
```

现在可以用 **`lein run`** 在本地执行任务，提供输入和输出文件：

```
$ lein run -m cookbook.etl.Main samples/books/books.json samples/books/output

# Or, on a Hadoop cluster
$ lein uberjar
$ hadoop jar target/cookbook-standalone.jar cookbook.etl.Main \
  books.json books.csv
```

samples/books/output/part-00000 中的结果是：

```
Ryan,Luke
```

# 讨论

虽然写一个脚本将 JSON 转换成 CSV 很容易，但要让这个脚本在许多机器上运行，就需要大量工作。用 Cascalog 来编写转换脚本，运行在本地模式或分布式模式时几乎不需要修改。

前面的小例子中有许多新概念和语法，让我们一条一条地分析一下。

在这个实例中，数据基本上按顺序流过函数。第一行用 `defmain` 宏（来自 Cascalog）定义了一个带有 `-main` 函数的类，让你在 Hadoop 上执行查询。在这个例子中，带有 `-main` 函数的类名为 `Main`，但这不是必须的。`defmain` 允许你在同一个文件中创建多个支持 Hadoop 的查询：

```
(defmain Main [in out & args]
```

在 `Main` 函数中是一个 Cascalog 操作符，`?<-`[1]，它定义并执行一个查询：

```
(?<-
```

注 1：虽然查询看起来“像”普通的 Clojure，但它们实际上是 DSL。如果你不熟悉 Cascalog 查询，可以从 Nathan Marz 的文章 Introducing Cascalog（http://nathanmarz.com/blog/introducing-cascalog-a-clojure-based-query-language-for-hado.html）中了解更多的信息。

这个操作接受一个输出位置 [ 在 Cascalog 中称为“龙头”（tap）]，一个结果向量，以及一系列逻辑谓词。下一行是目的地，输出将写入的位置。同样的函数被用于创建输入和输出龙头：

```
(hfs-textline out :sinkmode :replace)
```

这个例子用了 `hfs-textline`，但还有许多其他龙头。你甚至可以自己写。

在你的输出龙头中使用 `:sinkmode :replace`，Cascalog 将覆盖所有原有的输出。如果你要重新运行一个查询并对它进行调试，这是有帮助的。否则，你不得不在每次重新运行时，删除输出文件。

这是一个列表，包含该查询应该返回的所有逻辑变量：

```
[?out-csv]
```

在这个例子中，这些是将要存放到输出位置的逻辑变量。Cascalog 知道它们是特殊的逻辑变量，因为它们的名字以 ? 或 ! 开头。

在理解逻辑变量时，可以将它们看成是包含所有可能有效的值。如果你添加谓词，要么引入了新的逻辑变量，它们（预期会）链接到原有的逻辑变量，要么对原有的逻辑变量添加了约束。

下一行定义了输入龙头。JSON 数据结构将从 `in` 指定的位置，每次读入一行。每一行都会存储在 `?in-json` 逻辑变量中，它将流过余下的逻辑谓词：

```
((hfs-textline in) ?in-json)
```

`read-str` 将 `?in-json` 中的 JSON 字符串解析为一个散列映射表，保存在 `?book-map` 中：

```
(json/read-str ?in-json ?book-map)
```

现在你将 authors 从映射表中取出，并将该向量保存到它自己的逻辑变量中。Cascalog 假定向量输出意味着绑定多个逻辑变量。要绕过 Cascalog 的假定，就把输出包装在一个额外的向量中，供 Cascalog 使用：

```
(get-vec ?book-map "authors" ?authors)
```

最后，利用 `vec->csv` 函数，将作者向量转换成合法的 CSV。这一行将为逻辑变量 `?out-csv` 产生值，它的名字曾出现在前面的输出行中，因此该查询将产生输出：

```
(vec->csv ?authors ?out-csv)))
```

对于构建抽取转换加载（ETL）管道，Cascalog 是一个很好的工具。它让你有更多的时间来思考数据，花更少的时间来考虑读取文件、分布式工作和管理依赖关系。在编写你自己的 ETL 管道时，遵循下面的步骤或许有帮助。

(1) 确定输入格式。
(2) 确定输出格式。
(3) 从输入格式开始工作，追踪每一步当前的格式。

## 参阅

- Ian Rumford 的博客文章“Using Cascalog for Extract Transform and Load”（http://ianrumford.github.io/blog/2012/09/29/using-cascalog-for-extract-transform-and-load/）。
- `core.logic`（https://github.com/clojure/core.logic），Clojure 的逻辑编程库。

# 9.3 聚合大型文件

作者：Alex Robbins

## 问题

需要从一些几个 T 的日志文件中，产生聚合的统计结果。例如，对于简单的输入日志文件（`<date>,<URL>,<USER-ID>`）：

```
20130512020202,/,11
20130512020412,/,23
20130512030143,/post/clojure,11
20130512040256,/post/datomic,23
20130512050910,/post/clojure,11
20130512051012,/post/clojure,14
```

希望输出这样的聚合统计结果：

```
{
"URL"  {"/"              2
        "/post/datomic"  1
        "/post/clojure"  3}
"User" {"23" 2
        "11" 3
        "14" 1}
"Day"  {"20130512" 6}
}
```

## 解决方案

Cascalog 允许你编写分布式处理任务，运行在本地或 Hadoop 集群上。

要继续这个实例，请复制 Cascalog samples GitHub 代码库（https://github.com/clojure-cookbook/cascalog-samples），并签出 `aggregation-begin` 分支。这将提供一个基本的 Cascalog 项目，就像 9.2 节“用抽取转换加载（ETL）管道来处理数据”中创建的一样：

```
$ git clone https://github.com/clojure-cookbook/cascalog-samples.git
$ cd cascalog-samples
$ git checkout aggregation-begin
```

现在将 `[cascalog/cascalog-more-taps "2.0.0"]` 添加到项目依赖关系中，将 `cookbook.aggregation` 设置为 AOT 编译。project.clj 应该看起来像这样：

```
(defproject cookbook "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [cascalog "2.0.0"]
                 [cascalog/cascalog-more-taps "2.0.0"]
                 [org.clojure/data.json "0.2.2"]]
  :profiles {:dev  {:dependencies [[org.apache.hadoop/hadoop-core "1.1.2"]]}}
  :aot [cookbook.etl
        cookbook.aggregation])
```

创建文件 src/cookbook/aggregation.clj，在其中添加聚合查询：

```
(ns cookbook.aggregation
  (:require [cascalog.api :refer :all]
            [cascalog.more-taps :refer [hfs-delimited]]))

(defn init-aggregate-stats [date url user]
  (let [day (.substring date 0 8)]
    {"URL" {url 1}
     "User" {user 1}
     "Day" {date 1}}))

(def combine-aggregate-stats
  (partial merge-with (partial merge-with +)))

(defparallelagg aggregate-stats
  :init-var    #'init-aggregate-stats
  :combine-var #'combine-aggregate-stats)

(defmain Main [in out & args]
  (?<-
  (hfs-textline out :sinkmode :replace)
  [?out]
```

```
  ((hfs-delimited in :delimiter ",") ?date ?url ?user)
  (aggregate-stats ?date ?url ?user :> ?out)))
```

在 samples/posts/posts.csv 文件中添加一些样本数据：

```
20130512020202,/,11
20130512020412,/,23
20130512030143,/post/clojure,11
20130512040256,/post/datomic,23
20130512050910,/post/clojure,11
20130512051012,/post/clojure,14
```

这个解决方案的全部内容都可以在 Cascalog samples 代码库（https://github.com/clojure-cookbook/cascalog-samples）的 `aggregation-complete` 分支中找到。

要获取这个工作项目的拷贝和样本数据，请签出该分支。

```
$ git checkout aggregation-complete
```

现在你可以在本地执行该任务：

```
$ lein run -m cookbook.aggregation.Main \
  samples/posts/posts.csv samples/posts/output

# Or, on a Hadoop cluster
$ lein uberjar
$ hadoop jar target/cookbook-standalone.jar \
             cookbook.aggregation.Main \
             samples/posts/posts.csv samples/posts/output
```

samples/posts/output/part-00000 中的结果根据可读性进行了格式化，像这样：

```
{
"URL"  {"/"             2
        "/post/datomic" 1
        "/post/clojure" 3}
"User" {"23" 2
        "11" 3
        "14" 1}
"Day"  {"20130512" 6}
}
```

# 讨论

Cascalog 使得快速产生聚合统计数字变得很容易。对于某些映射归约（MapReduce）框架，聚合统计数字可能有点麻烦。一般来说，映射归约任务的映射阶段会在集群上很好地分布，但归约阶段常常分布得不那么好。例如，无经验的人来设计聚合算法，会让所有聚合工作在一台归约机器上执行。如果所有聚合在一个节点上完成，那么在归约阶段，2000 台

计算机集群就会像 1 台计算机一样慢。

在开始编写自己的聚合算法之前，请先查看 `cascalog.logic.ops` 的源代码。这个命名空间有许多有用的函数，可能已经完成了你想做的事。

在我们的例子中，目标是计算每个 URL 出现的次数。要创建最终的映射表，所有的 URL 需要最终集中到一个归约器上。没有经验的程序实现会对所有元组使用一个聚合算法。这意味着只在一个节点上完成所有的工作，计算所花的时间就像在一台计算机上一样。

解决方案是利用 Hadoop 的结合器（combiner）函数。结合器函数在输出发送到归约器之前，针对映射阶段的结果执行。最重要的是，结合器运行在映射器的节点上。这意味着结合器的工作分布在整个集群上，就像映射的工作一样。如果大多数工作都在映射和结合器的阶段完成，归约的阶段就会立刻完成。Cascalog 让它变得非常容易。许多内建的 Cascalog 函数都在背后用到了结合器，所以你甚至不用试，就能写出高度优化的查询。你甚至可以用 `defparallelagg` 宏，编写自己的函数来利用结合器。

> Cascalog 常常处理 var，而不是那些 var 的值。例如，调用 `defparallelagg` 时接受引用的参数。`#'` 语法意味着 var 被传递，而不是 var 引用的值。Cascalog 传递这些 var 而不是值，这样它就不必将函数序列化，再将这些函数传递给映射器或归约器。它只要传递 var 的名称，该名称将在远程执行环境中查找。这意味着你不能为 Cascalog 工作流的某些部分动态地构建函数。大多数函数需要绑定到 var。

`defparallelagg` 开始有点令人迷惑，但它有通过编写查询，利用结合器的能力，因此值得学习。你需要提供两个 var，指向 `defparallelagg` 调用的函数：`init-var` 和 `combine-var`。请注意，这两个参数都是作为 var 传递，而不是函数值，因此需要在名称前面加上 `#'`。`init-var` 函数需要接收输入数据，改变它的格式，以便容易被 `combine-var` 函数处理。在这个例子中，解决方案将数据变成包含一些映射表的映射表，能够很容易合并。合并映射表是编写并行聚合器的一种简单方法。`combine-var` 函数必须是可传递的、可关联的。该函数被调用接收 `init-var` 函数的两个输出实例。返回值将作为参数，传递给稍后的 `combine-var` 函数调用。输出对将被合并，直到只剩下一个输出，这就是最终输出。

下面详细解释一下查询。

首先，请求你需要的 Cascalog 函数：

```
(ns cookbook.aggregation
  (:require [cascalog.api :refer :all]
            [cascalog.more-taps :refer [hfs-delimited]]))
```

然后定义一个函数 `init-aggregate-stats`，它接受一个日期、URL 和用户，返回一个包含

映射表的映射表。第二级别的映射含有与被观察的值所对应的键。这是 init 函数，它接受每一行记录，准备好进行聚合：

```
(defn init-aggregate-stats [date url user]
  (let [day (.substring date 0 8)]
    {"URL"  {url 1}
     "User" {user 1}
     "Day"  {date 1}}))
```

combine-aggregate-stats 函数接受 init-aggregate-stats 函数对于所有输入数据的输出，并结合在一起。这个函数会反复调用，结合 init-aggregate-stats 函数的输出和它自己的输出。它的输出应该和输入格式一样，因为这个函数将对两个输出进行调用，直到只剩下一个输出数据。这个函数合并嵌套的映射表，将相同的键对应的值加起来。

```
(def combine-aggregate-stats
  (partial merge-with (partial merge-with +)))
```

aggregate-stats 接受前面两个函数，将它们转化成 Cascalog 的并行聚合操作。请注意，你传递的是 var，而不是函数本身：

```
(defparallelagg aggregate-stats
  :init-var    #'init-aggregate-stats
  :combine-var #'combine-aggregate-stats)
```

最后，建立 Main 来定义并执行查询，该查询对 in 中所有输入调用 aggregate-stats 操作，结果写入 out：

```
(defmain Main [in out & args]
  ;; 这定义并执行一次 Cascalog 查询
  (?<-
    ;; 设置输出路径
    (hfs-textline out :sinkmode :replace)
    ;; 定义哪些逻辑变量将输出
    [?out]
    ;; 设置输入路径，定义绑定到输入的逻辑变量
    ((hfs-delimited in) ?date ?url ?user)
    ;; 执行聚合操作
    (aggregate-stats ?date ?url ?user :> ?out)))
```

如果你想计算的聚合不能用 defparallelagg 来定义，Cascalog 还提供了其他一些方法来定义聚合。但是，其中许多没有使用结合器，可能导致大多数计算发生在少数归约器上。计算也许能完成，但丧失了分布式计算的许多好处。请查看 cascalog.logic.ops 的源代码，看看有哪些不同的方法，以及如何使用它们。

## 参阅

- cascalog.logic.ops 的源代码（https://github.com/nathanmarz/cascalog/blob/develop/cascalog-core/src/clj/cascalog/logic/ops.clj），该命名空间包含了许多预定义的操作（包括聚合器）。

# 9.4 测试Cascalog工作流

作者：Alex Robbins

## 问题

你喜欢测试你的代码，喜欢编写 Cascalog 任务，但讨厌试着去测试你的 Cascalog 任务。

## 解决方案

Midje-Cascalog（https://github.com/nathanmarz/cascalog/tree/develop/midje-cascalog）提供了少量的附加功能，让编写 Cascalog 任务的测试变得相当容易。要继续这个实例，请复制 Cascalog samples GitHub 代码库（https://github.com/clojure-cookbook/cascalog-samples）并签出 `testing-begin` 分支。这将提供一个基本的 Cascalog 项目，就像 9.2 节“用抽取转换加载（ETL）管道来处理数据”中创建的一样：

接下来将 Midje 插件和 Midje-Cascalog 依赖关系添加到 project.clj 文件的 `:dev` 特性描述。`:profiles` 键看起来应该像这样：

```
(defproject cookbook "0.1.0-SNAPSHOT"
  ;; ...
  :profiles {:dev {:dependencies [[org.apache.hadoop/hadoop-core "1.1.2"]
                                  [cascalog/midje-cascalog "2.0.0"]]
                   :plugins [[lein-midje "3.1.1"]]}})
```

在 src/cookbook/test_me.clj 中创建一个简单的查询，针对它编写测试：

```
(ns cookbook.test-me
  (:require [cascalog.api :refer :all]))

(defn capitalize [s]
  (.toUpperCase s))

(defn capitalize-authors-query [author-path]
  (<- [?capitalized-author]
    ((hfs-textline author-path) ?author)
    (capitalize ?author :> ?capitalized-author)))
```

现在，你可以在 test/cookbook/test_me_test.clj 中，为这个查询写一个测试：

```
(ns cookbook.test-me-test
  (:require [cookbook.midje-cascalog :refer :all]
            [midje
              [sweet :refer :all]
              [cascalog :refer :all]]))

(fact "Query should return capitalized versions of the input names."
```

```
(capitalize-authors-query :author-path) => (produces [["LUKE VANDERHART"]
                                                      ["RYAN NEUFELD"]])
(provided
  (hfs-textline :author-path) => [["Luke Vanderhart"]
                                  ["Ryan Neufeld"]]))
```

这个解决方案的全部内容都在 Cascalog samples 代码库（https://github.com/clojure-cookbook/cascalog-samples）的 `testing-complete` 分支中。

要获取这个工作项目的副本和样本数据，请签出该分支。

```
$ git checkout testing-complete
```

最后，用 **`lein midje`** 运行测试：

```
$ lein midje
2013-11-09 12:19:27.844 java[3620:1703] Unable to load realm info from
  SCDynamicStore
All checks (1) succeeded.
```

## 讨论

单元测试是重要的“软件手艺”。但是，不夸张地说，单元测试 Hadoop 工作流是困难的。大多数分布式计算开发都是通过试错来完成的，只经过有限的手工测试，然后就认为工作流“足够好”了，投入生产环境使用。你不应该让代码的质量滑坡，但测试分布式代码可能比较难。Midje-Cascalog 让测试 Cascalog 工作流的不同部分变得容易，因为它使得模仿子查询的结果变得极其简单。

在前面概述的解决方案中，你打算测试一个简单的查询。它从输入路径中读取一些行，将它们转为大写，然后输出。通常，你需要确保部分测试将一些测试数据写入文件，在测试中引用该文件，然后清理并删除该文件。但如果使用 Midje-Cascalog，你将模拟 `hfs-textline` 调用。

`fact` 是 Midje 库提供的，它本身也很值得学习。它是 `clojure.test` 的 `deftest` 的替代方案。在这里，你将测试声明为一个调用，跟着是一个箭头，然后是 `produces` 函数。`produces` 让你将查询的结果写成一个包含向量的向量。建立好测试后，你用 `provided` 来描述你想模拟的函数。这让你只测试待测函数，而不是它依赖的那些函数。测试 Cascalog 工作流和测试应用的其他部分一样重要。有了 Midje-Cascalog，这确实可以做到。

## 参阅

- 10.2 节“用 Midje 测试”。
- GitHub 上 的 Midje-Cascalog 文 档（https://github.com/nathanmarz/cascalog/tree/develop/midje-cascalog）。

# 9.5 设置Cascalog任务的检查点

作者：Alex Robbins

## 问题

长期运行的 Cascalog 任务抛出了错误，然后需要完全重启。你浪费了时间，等待处于工作流后期的一些问题重新执行前面的步骤。

## 解决方案

Cascalog Checkpoint（https://github.com/nathanmarz/cascalog-contrib/tree/master/cascalog.checkpoint）是一个优秀的库，提供了在 Cascalog 任务中添加检查点的能力。如果一步失败，任务将从那一步重新开始，而不是从头再来。

在已有的 Cascalog 项目中，例如 9.2 节“用抽取转换加载（ETL）管道来处理数据”中创建的项目，在项目依赖关系中添加 `[cascalog/ cascalog-checkpoint "1.10.2"]`，将 `cookbook.checkpoint` 命名空间设置为 AOT 编译的。

然后用 Cascalog Checkpoint 的 `workflow` 宏来设置你的任务。假设一个四步的任务看起来是这样的：

```
(ns cookbook.checkpoint
  (:require [cascalog.api :refer :all]
            [cascalog.checkpoint :refer [workflow]]))

(defmain Main [in-path out-path & args]
  (workflow ["/tmp/log-parsing"]
    step-1 ([:temp-dirs parsed-logs-path]
            (parse-logs in-path parsed-logs-path))
    step-2 ([:temp-dirs [min-path max-path]]
            (get-min parsed-logs-path min-path)
            (get-max parsed-logs-path max-path))
    step-3 ([:deps step-1 :temp-dirs log-sample-path]
            (sample-logs parsed-logs-path log-sample-path))
    step-4 ([:deps :all]
            (summary parsed-logs-path
                     min-path
                     max-path
                     log-sample-path
                     out-path))))
```

## 讨论

Cascalog 任务常常需要几个小时来运行。有些事情比打字错误更令人沮丧，它们会在最后的步骤中破坏任务，而任务已经运行了整个周末。Cascalog Checkpoint 提供了 `workflow` 宏，这让你从最后成功的步骤开始重启任务。

`workflow` 宏的第一个参数 `checkpoint-dir` 是一个向量，包含一个路径，用于存放临时文件。每一步的输出被临时存放在该路径的目录中，还有一些文件来追踪哪些步骤已经成功完成。

在第一个参数之后，`workflow` 需要几对步骤名称和步骤定义。一个步骤定义是一个选项向量，跟着该步骤包含的 Cascalog 查询，数目不限。例如：

```
step-3 ([:deps step-1 :temp-dirs [log-sample-path log-other-sample-path]]
       (sample-logs parsed-logs-path log-sample-path)
       (other-sample-logs parsed-logs-path log-other-sample-path))
```

这个步骤定义确定了 `step-3`。它依赖于 `step-1`，所以它在 `step-1` 完成后才会运行。这个步骤为它的查询创建了两个临时目录。`:deps` 和 `:temp-dirs` 既可以是一个符号，也可以是一个包含符号的向量，或者可以省略。在选项向量之后，你可以包含一个或多个 Cascalog 查询。在这个例子中，有两个查询。

`:deps` 可以接受不同的值。`:last` 是默认的值，让该步骤依赖于它前面的步骤。`:all` 让该步骤依赖于前面定义的所有步骤。提供一个符号，或包含符号的向量，让该步骤依赖于特定的一个或一组步骤。步骤只有在它依赖的步骤完成后，才会执行。如果几个步骤的依赖关系都已满足，它们会并行执行。

提供给 `:temp-dirs` 的每个符号都会被转变成临时路径下的一个目录。后来的步骤将使用这些目录，读取早期步骤的输出数据。在工作流全部成功完成时，这些目录会被清理。在此之前，这些目录将保留不同步骤的输出，以便让工作流能够从最后未完成的步骤开始继续执行。

> 如果你希望重启已经成功完成的步骤，请删除 <checkpoint-dir>/<step-name> 中的文件。如果你需要删除或修改其中的数据，步骤定义中的 `:temp-dirs` 可以在 <checkpoint-dir>/data/<temp-dir> 中找到。

处理错误的另一个方法是为 Cascalog 查询提供错误陷井。Cascalog 会将导致查询错误的输入元组放入错误陷井（进行不同的处理，或倒出供手工检查）。有了错误陷井，一些不良输入就不会让整个工作流停下来。

为 Cascalog 任务设置检查点需要在一开始多做一点工作，但它会为你节省大量的时间。事情会出错。集群会宕机。你会发现打字错误和边界情况。能够从最后完成的步骤开始重启任务，而不是每次等待运行所有的步骤，这是很好的。

## 参阅

- GitHub 上的 `cascalog.checkpoint` 项目页面（https://github.com/nathanmarz/cascalog-contrib/tree/master/cascalog.checkpoint）。

# 9.6 解释Cascalog查询

作者：Alex Robbins

## 问题

Cascalog 任务运行得很慢，你不能确定原因。

## 解决方案

利用 `cascalog.api/explain` 函数，打印出查询的 DOT 文件。要继续本实例，可以在已有的项目中启动 REPL，例如 9.2 节“用抽取转换加载（ETL）管道来处理数据”中创建的项目：

```
(require '[cascalog.api :refer [explain <-]])

(explain "slow-query.dot" (<- [?a ?b] ([[1 2]] ?a ?b)))
```

接下来，你需要查看 DOT 文件。有许多种方式查看，但最容易的方式可能是利用 `dot`，它是 Graphviz 的工具之一，将 DOT 文件转成 PNG 或 GIF：

```
$ dot -Tpng -oslow-query.png slow-query.dot
```

现在打开 slow-query.png（如图 9-1 所示），看看查询的图示。

## 讨论

Cascalog 工作流编译成了 Cascading 工作流。Cascading（http://www.cascading.org/）是一个 Java 库，它包装了 Hadoop，提供了基于流的管道抽象。DOT 文件中的查询图将包含一些不同的 Cascading 元素作为结点。

这里的 `explain` 函数类似于许多 SQL 实现中的 EXPLAIN 命令。`explain` 导致 Cascalog 打印出查询计划。就像 SQL 的 EXPLAIN 的输出一样，你必须去理解看到的东西。

要检查的最重要的事情，就是查询的基本流程确实符合你的期望。确保不是在重复运行查询的某些部分。Cascalog 让查询复用变得很容易，但你常常希望运行查询，保存结果，然后引用从其他查询保存的结果，而不是每次用到它的输出时都运行一次。

你也可以尝试匹配查询计划中的阶段和运行的任务。这有些麻烦，因为阶段不会刚好对应于输出图。但是如果成功，就能追踪到慢的阶段。

一般来说，要保持 Cascalog 查询速度，就要确保使用了集群中的所有节点。这意味着让工作保持为平均规模的小单元。如果一个映射器的输入花的时间是另外 40 个输入的 1000

倍，那么整个任务将等待那一个映射器完成工作。将那个长时间的映射任务切分成 1000 个较小的任务，将使整个任务快很多，因为它能够分布到整个集群，而不是在单个节点上运行。很容易发生在一个归约器中终止几乎整个任务的情况。如果几乎所有的归约器都已完成工作，任务在等待一两个归约器，那么很容易在 Hadoop 任务追踪器中看到。要解决这个问题，就要在映射阶段利用聚合器完成尽可能多的归约工作，然后确保剩下的归约工作没有堆积到少数的归约器上。

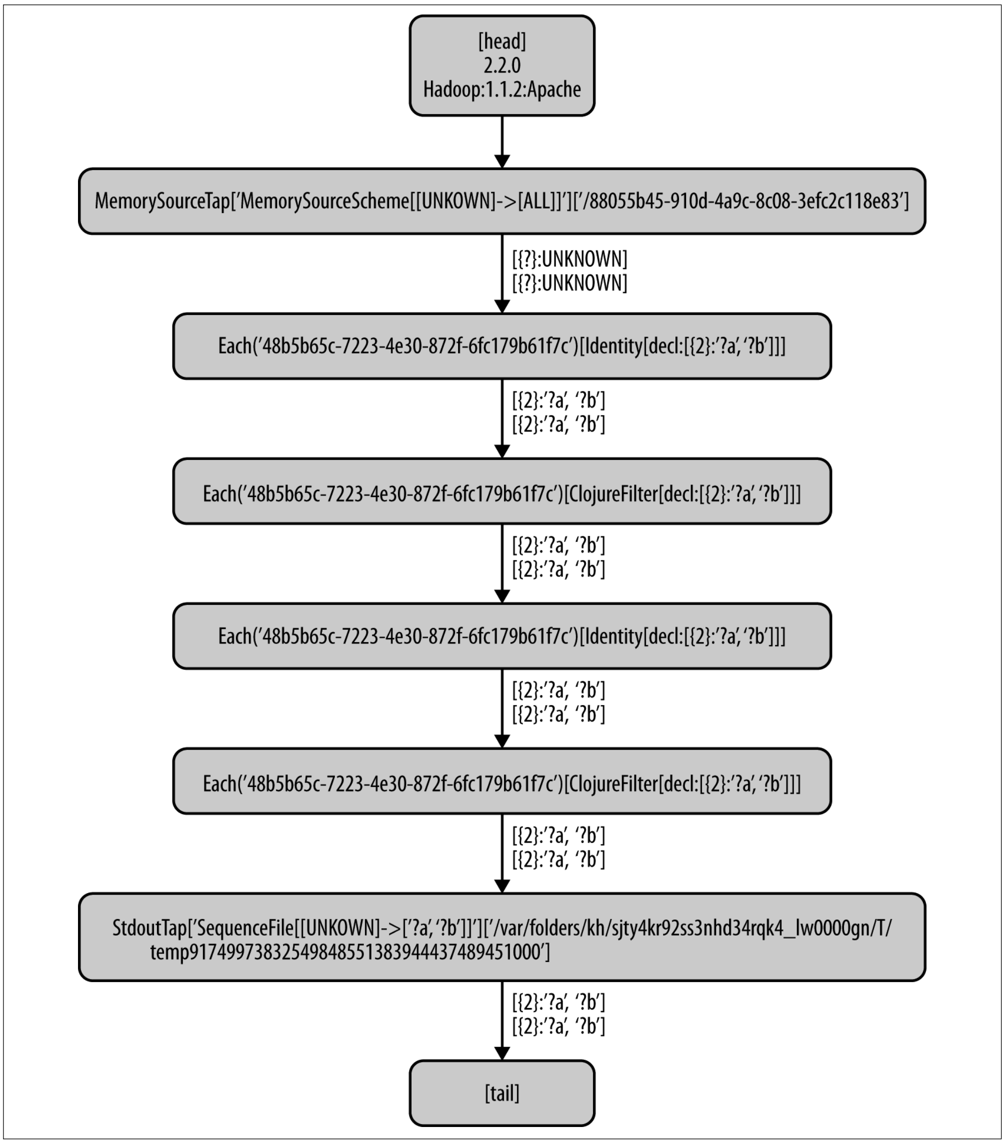



图 9-1：slow-query.png

## 参阅

- 9.3 节“聚合大型文件”。
- Cascalog 维基页面上的“Cascading Flow Visualization”（https://github.com/nathanmarz/cascalog/wiki/Cascading-Flow-visualization）。

# 9.7 在Elastic MapReduce上运行Cascalog任务

作者：Alex Robbins

## 问题

你有大量数据要处理，但没有 Hadoop 集群。

## 解决方案

亚马逊的 Elastic MapReduce（http://aws.amazon.com/elasticmapreduce/，EMR）提供了按需计算的 Hadoop 集群。要使用 EMR，你需要一个 Amazon Web Services 账户（http://aws.amazon.com/cn/）。

首先，像平时一样编写一个 Cascalog 任务。本章有一些实例能够帮助你创建完整的 Cascalog 任务。如果没有自己的任务，也可以复制 9.2 节“用抽取转换加载（ETL）管道来处理数据”：

```
$ git clone https://github.com/clojure-cookbook/cascalog-samples.git
$ cd cascalog-samples
$ git checkout etl-sample
```

有了 Cascalog 项目后，打包成一个 uberjar：

```
$ lein compile
$ lein uberjar
```

接下来，将生成的 JAR 上传到 S3（如果你按照 ETL 的例子做，就是 target/cookbook-0.1.0-SNAPSHOT-standalone.jar）。若你从未上传文件到 S3，可查阅 S3 文档“Create a Bucket”（http://docs.aws.amazon.com/AmazonS3/latest/gsg/CreatingABucket.html）和“Adding an Object to a Bucket”（http://docs.aws.amazon.com/AmazonS3/latest/gsg/PuttingAnObjectInABucket.html）。重复这一步骤，以上传你的输入数据。在这一过程中，请留意 JAR 的路径以及输入数据的位置。

要创建你的 MapReduce 任务，请访问 https://console.aws.amazon.com/elasticmapreduce/，选择“Create New Job Flow”（图 9-2）在进入新任务流向导后，选择“Custom JAR”任务类

型。选择“Continue”并输入你的 JAR 的位置和参数。“JAR Location”就是前面通知你的 S3 路径。“JAR Arguments”是执行你的 JAR 时要传入的所有参数。例如，使用 Cascalog samples 代码库中的例子时，参数就是要执行的完整限定的类名，cookbook.etl.Main，一个包含输入数据的 s3n:// URI，和一个输出数据的 s3n:// URI。

接下来几个向导窗口让你为任务指定附加的配置选项。选择“Continue”，直到到达复查（REVIEW）阶段，然后开始你的任务。

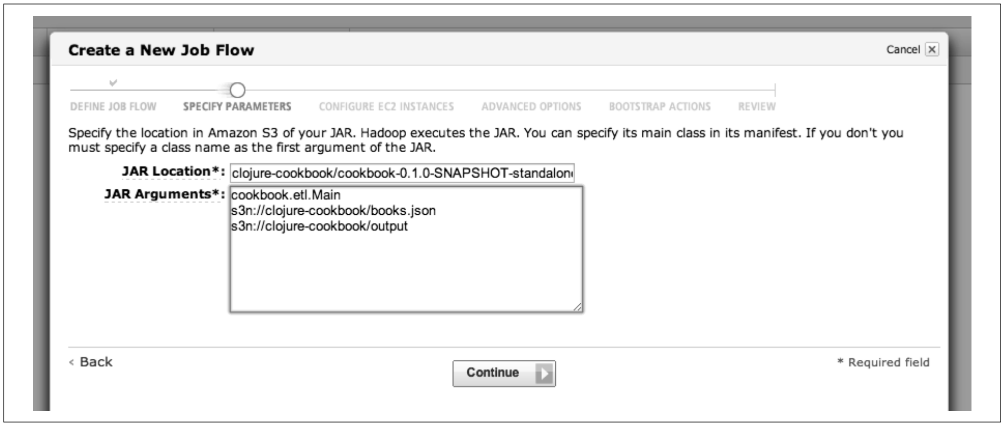



图 9-2：在新任务向导中指定参数

在任务运行完后，你应该能够从 S3 获取结果。Elastic MapReduce 允许你设置一个日志路径，如果任务没有像预期那样完成，它可以辅助调试。

## 讨论

如果你有大的 Cascalog 任务，又不是经常需要运行它们，亚马逊的 EMR 是一个很好的解决方案。维护自己的 Hadoop 集群需要大量的时间和金钱。如果你能让集群保持很忙，这是很好的投资。如果每月只需要用几次，最好还是用 EMR 的按需计算 Hadoop 集群。

## 参阅

- 9.2 节“用抽取转换加载（ETL）管道来处理数据”，了解创建一个简单的 Cascalog 任务。
- 亚马逊的“Launch a Custom JAR Cluster”文档（http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-launch-custom-jar-cli.html）。

# 第10章

# 测试

## 10.0 简介

今天对代码正确有信心是一回事，但一周后你是什么感觉？一个月后呢？一年后呢？在你离开很久之后呢？为了找到这种信心，我们为代码编写测试。写得好的测试套件对你自己和之后的开发者宣称：“只要这个测试通过，不论是现在还是将来，这就是应用程序工作的方式”。

除了测试，一些其他的工具最近也在 Clojure 中出现，目的是改进程序的可靠性。通常，它们专注于验证数据是否符合预期，防止程序收到不知如何处理的输入。这些工具各式各样，从简单的前置条件，到可选的静态类型和编译时的代数类型分析。

必须承认，现在测试在 Clojure 社区中是热点话题。人们开始质疑这些测试是否值得，或者是否有更好的方式来考虑程序验证。近几年，一些技术，如 REPL 驱动的开发、基于属性的测试和可选的类型，已经异军突起，填补了测试领域中大家看到的空白。

本章介绍了以上所有技术。尽管我们喜爱挑战极限，但通常好的老式单元测试套件仍是最佳选择。同时，随着我们创建了越来越多的庞大应用，简单的单元测试有时会不够，这一点很清楚。我们希望不论你的技能水平或关注点如何，都能在本章中找到新的工具，添加到你的测试武器库中。

# 10.1 单元测试

作者：Daniel Gregoire

## 问题

希望测试 Clojure 代码的独立单元。

## 解决方案

在 clojure.test 命名空间中，Clojure 包含了一个单元测试框架。它提供了一些方式来命名和分组测试，提供断言，报告结果，以及精心安排测试套件。为了示范，假设有一个 capitalize-entries 函数，将映射表中的值转换成首字母大写。要测试该函数，就用 clojure.test/deftest 定义一个测试：

```
;; 命名空间 com.example.core 中的一个函数
(defn capitalize-entries
  "Returns a new map with values for keys 'ks' in the map 'm' capitalized."
  [m & ks]
  (reduce (fn [m k] (update-in m [k] clojure.string/capitalize)) m ks))

;; The corresponding test in namespace com.example.core-test
(require '[clojure.test :refer :all])

;; 在真正的测试命名空间，你也会 :refer 所有的目标命名空间
;; (require '[com.example.core :refer :all])

(deftest test-capitalize-entries
  (let [employee {:last-name "smith"
                  :job-title "engineer"
                  :level 5
                  :office "seattle"}]
    ;; Passes
    (is (= (capitalize-entries employee :job-title :last-name)
           {:job-title "Engineer"
            :last-name "Smith"
            :office "seattle"
            :level 5}))
    ;; Fails
    (is (= (capitalize-entries employee :office)
           {}))))
```

用 clojure.test/run-tests 函数来运行测试：

```
(run-tests)
;; -> {:type :summary, :pass 1, :test 1, :error 0, :fail 1}
;; *out*
;; Testing user
;;
```

```
;; FAIL in (test-capitalize-entries) (NO_SOURCE_FILE:13)
;; expected: (= (capitalize-entries employee :office) {})
;;   actual: (not (= {:last-name "smith", :office "Seattle",
;;                    :level 5, :job-title "engineer"} {}))
;;
;; Ran 1 tests containing 2 assertions.
;; 1 failures, 0 errors.
```

# 讨论

前面的例子对 clojure.test 提供的单元测试功能只是浅尝辄止。让我们以从下向上的方式，看看它的其他功能。

首先，可以改进断言失败时的报告，提供第二个参数，解释该断言的测试意图。如果运行这个测试，就会看到一段扩展的描述，说明代码预期的行为：

```
(is (= (capitalize-entries {:office "space"} :office) {})
    "The employee's office entry should be capitalized.")
;; -> false
;; * out*
;; FAIL in clojure.lang.PersistentList$EmptyList@1 (NO_SOURCE_FILE:1)
;; The employee's office entry should be capitalized.
;; expected: (= (capitalize-entries {:office "space"} :office) {})
;;   actual: (not (= {:office "Space"} {}))
```

为了全面测试像 capitalize-entries 这样的函数，需要考虑几种用例。要更精确地测试大量的类似用例，请使用 clojure.test/are 宏：

```
(deftest test-capitalize-entries
  (let [employee {:last-name "smith"
                  :job-title "engineer"
                  :level 5
                  :office "seattle"}]
    (are [ks m] (= (apply capitalize-entries employee ks) m)
         [] employee
         [:not-a-key] employee
         [:job-title] {:job-title "Engineer"
                       :last-name "smith"
                       :level 5
                       :office "seattle"}
         [:last-name :office] {:last-name "Smith"
                               :office "Seattle"
                               :level 5
                               :job-title "engineer"})))
```

are 的前两个参数建立起了一种测试模式：给定一系列的键 ks 以及一个映射表 m，用这些键和原来的 employee 映射表调用 capitalize-entries，断言返回值等于 m。

用描述式语法编写多个用例，更容易捕捉错误和未处理的边界情况，例如对于前面测试的 [:not-a-key] employee 断言对，会抛出 NullPointerException。

不像其他流行动态语言的测试框架，Clojure 内建的断言最少而且简单。`is` 和 `are` 宏检查测试表达式是否为“真”（也就是说，如果这些表达式返回的既不是 `false` 也不是 `nil`，测试就通过）。此外，也可以检查 `thrown?` 或 `thrown-with-msg?`，测试某个预期的 `java.lang.Throwable`（错误或异常）：

```
(is (thrown? IndexOutOfBoundsException (nth [] 1)))
```

在单个断言的层面之上，`clojure.test` 也提供一些机制，在测试运行之前或之后调用一些函数。在 `test-capitalize-entries` 测试中，我们定义了一个随意的 `employee` 映射表来测试，但也可以注册一个数据加载函数作为“测试装置”（fixture），读入外部数据，在多个测试中共享。Clojure.test/use-fixfures 的多重方法允许 Clojure 注册函数可以在测试前后被调用，或在整个命名空间的测试套件前后调用。下面的例子定义并注册了三个测试装置函数：

```
(require '[clojure.edn :as edn])

(def test-data (atom nil))

;; 假定你有一个 test-data.edn 文件……
(defn load-data "Read a Clojure map from test data in a file."
  [test-fn]
  (reset! test-data (edn/read-string (slurp "test-data.edn")))
  (test-fn))

(defn add-test-id "Add a unique id to the data before each test."
  [test-fn]
  (swap! test-data assoc :id (java.util.UUID/randomUUID))
  (test-fn))

(defn inc-count "Increment a counter in the data after each test runs."
  [test-fn]
  (test-fn)
  (swap! test-data update-in [:count] (fnil inc 0)))

(use-fixtures :once load-data)
(use-fixtures :each add-test-id inc-count)

;; Tests...
```

你可以认为测试装置函数构成了一个管道，每个测试作为一个参数通过它，在前面的例子中我们将测试称为 `test-fn`。以 `inc-count` 为例。这个测试装置的任务是调用 `test-fn` 函数，继续这个管道，然后增加计数（即“做一些工作”）。每个测试装置决定是在自己的工作之前还是之后调用 `test-fn`（请比较 `add-test-id` 函数和 `inc-count` 函数），而 `clojure.test/use-fixtures` 多重方法则实现控制，决定每个注册的测试装置函数对于命名空间中的所有测试只运行一次，还是对每个测试运行一次。

最后，在深刻理解了如何开发单个 Clojure 测试套件之后，你要考虑作为项目构建的一部

分，如何组织和运行这些套件，这一点很重要。尽管 Clojure 允许在代码集的任何地方定义函数的测试，但你应该将测试代码放在一个独立的目录，只在需要的时候（例如开发和测试的时候）才添加到 JVM 类路径中。根据被测试的命名空间来命名测试的命名空间会比较方便，这样位于 <project-root>/src/com/example/core.clj 的文件，命名空间是 `com.example.core`，对应的测试文件在 <project-root>/test/com/example/core_test.clj，命名空间是 `com.example.core-test`。要控制源代码的位置和测试代码的路径，以及是否包含在 JVM 类路径中，你应该使用 Leiningen（http://leiningen.org/）或 Maven（http://maven.apache.org/）这样的构建工具来组织你的项目。

在 Leiningen 中，测试代码的默认的目录是顶层的 <project-root>/test 文件夹，你可以在命令行用 `lein test` 来运行项目的测试。不带附加的参数，`lein test` 命令将执行项目中所有的测试：

```
$ lein test

lein test com.example.core-test
lein test com.example.util-test

Ran 10 tests containing 20 assertions.
0 failures, 0 errors.
```

要限制 Leiningen 运行测试的范围，就用 `:only` 选项，带上完全限定的命名空间或函数名称：

```
# 运行整个命名空间
$ lein test :only com.example.core-test

lein test com.example.core-test

Ran 5 tests containing 10 assertions.
0 failures, 0 errors.

# 运行某个特定的测试
$ lein test :only com.example.core-test/test-capitalize-entries

lein test com.example.core-test

Ran 1 tests containing 2 assertions.
0 failures, 0 errors.
```

## 参阅

- `clojure.test` 的 API 文档（http://clojure.github.io/clojure/clojure.test-api.html）包含了该单元测试框架的全部信息。
- 如果你使用的是 Maven，就用 `clojure-maven-plugin`（https://github.com/talios/clojure-

maven-plugin）来运行 Clojure 测试。这个插件将合并 Maven 标准的 src/test/clojure 目录中的 Clojure 测试，作为 Maven 构建生命周期中 `test` 阶段的一部分。你可以选择性地使用该插件的 `clojure:test-with-junit` 目标，为 Clojure 测试生成 JUnit 风格的报告输出。

# 10.2 用Midje测试

作者：Joseph Wilk

## 问题

希望对一个函数进行单元测试，它集成了外部依赖关系，如 HTTP 服务或数据库。

## 解决方案

使用 Midje（https://github.com/marick/Midje），它是一个测试框架，提供了一些方法来模拟函数，返回假结果。要继续本实例，用 `lein-try` 启动 REPL：

```
$ lein try midje clj-http
```

这是一个示例函数，它发出 HTTP 请求：

```
;; com.example.core 命名空间中的一个函数
(require '[clj-http.client :as http])

(defn github-profile [username]
  (let [response (http/get (str "https://api.github.com/users/" username))]
    (when (= (:status response) 200)
      (:body response))))

(github-profile "clojure-cookbook")
;; -> "{\"login\":\"clojure-cookbook\",\"id\":4176246, ...}"
```

要测试 `github-profile` 函数，就在对应的测试命名空间中，用 `midje.sweet/facts` 和 `midje.sweet/fact` 定义一个测试：

```
;; 在 com.example.core-test 命名空间中……
(require '[midje.sweet :refer :all])

(facts "about successful requests"
  (fact "returns the response body"
    (github-profile "clojure-cookbook") => ..body..
    (provided
      (http/get #"/users/clojure-cookbook") =>
        {:status 200 :body ..body..})))
```

## 讨论

在 Midje 中，`facts` 将一段描述和一组测试关联起来，而 `fact` 映射到你的测试。`fact` 中的断言形式为：

```
;; actual => expected

10 => 10 ; 这会通过
10 => 11 ; 这会失败
```

断言的行为与大多数测试框架有一点不同。在 `fact` 主体内，每个断言都得到检查，不论前面的断言是否失败。

Midje 只提供模拟（mock），不提供桩（stub）。`provided` 主体内指定的所有函数都必须被调用，测试才能通过。模拟与断言使用了同样的语法，但含意稍有不同：

```
;; <function call & arguments to match> => <return value of function>

(provided (+ 10 10) => 0)
```

注意，这里你不是要调用 `(+ 10 10)` 函数，而是设置了一个模式，这一点很重要。测试中出现的每个函数调用都得到检查，看看它是否匹配这个模式。如果确实匹配，Midje 不会调用该函数，而是返回 `0`。在如何匹配模拟函数和真正调用方面，用 `provided` 来定义模拟函数非常灵活。例如，在前面的解决方案中，使用了正则表达式。这个表达式告诉 Midje 模拟对 `http/get` 的调用，只要 URL 以 /users/clojure-cookbook 结尾。

```
;; 预期
(http/get #"/users/clojure-cookbook$")

;; 将匹配
(http/get "http://localhost:4001/users/clojure-cookbook")
;; 或
(http/get "https://api.github.com/users/clojure-cookbook")
```

Midje 提供了许多匹配形状的函数，可以用来匹配模拟函数的参数：

```
;; 匹配包含 1 的参数列表
(provided
  (http/get (contains [1])) => :result)

;; 匹配一个定制的 fn，它必定返回 true
(provided
  (http/get (as-checker (fn [x] (x == 10)))) => :result)

;; 匹配单个参数，可以是任何值
(provided
  (http/get anything) => :result)
```

在 REPL 中，你可以研究所有 Midje 的检查器：

```
(require 'midje.repl)
(doc midje-checkers)
;; *out*
;; -------------------------
;; midje.sweet/midje-checkers
;;
;;  (facts "about checkers"
;;    (f) => truthy
;;    (f) => falsey
;;    (f) => irrelevant ; or `anything`
;;    (f) => (exactly odd?) ; when you expect a particular function
;;    (f) => (roughly 10 0.1)
;;    (f) => (throws SomeException #"with message")
;;    (f) => (contains [1 2 3]) ; works with strings, maps, etc.
;;    (f) => (contains [1 2 3] :in-any-order :gaps-ok)
;;    (f) => (just [1 2 3])
;;    (f) => (has every? odd?)
;;    (f) => (nine-of odd?) ; must be exactly 9 odd values.
;;    (f) => (every-checker odd? (roughly 9)) ; both must be true
;;    (f) => (some-checker odd? (roughly 9))) ; one must be true
```

你可能已经注意到，在解决方案中，我们用了 `..body..` 而不是真正的响应。这就是 Midje 所谓的*元常数*（metaconstant）。

元常数是任何用两个点开始和结束的名字。除标识外，它没有其他属性。可以把它看成是假结果或占位符，我们不关心它的实际值，或者它可能被目前还不存在的内容替换掉。在我们的例子中，我们并不关心 `..body..` 是什么，我们只关心它是返回的内容。

要在原有的项目中添加 Midje，就在开发依赖关系中添加 `[midje "1.5.1"]`，在开发插件中添加 `[lein-midje "3.1.2"]`。你的 project.clj 应该看起来像这样：

```
(defproject example "1.0.0-SNAPSHOT"
  :profiles {:dev {:dependencies [[midje "1.5.1"]]
                   :plugins [[lein-midje "3.1.2"]]}})
```

Midje 提供了两种方式来运行测试：通过 REPL，就像你可能已经在做的那样，或者通过 Leiningen。Midje 实际上鼓励你通过 REPL，在开发的时候运行所有测试。运行测试有一种很有用的方法，就是用 `midje.repl/autotest` 函数。它不断轮询文件系统，寻找项目中的变更。如果检测到这些变化，就会自动重新运行相关的测试：

```
(require '[midje.repl :as midje])

(midje/autotest) ; Start auto-testing

;; Other options are...
(midje/autotest :pause)
(midje/autotest :resume)
(midje/autotest :stop)
```

有了 Midje，你还可以在 REPL 中做许多事情。要了解更多信息，就在 REPL 中运行 **`(doc midje-repl)`**，阅读 `midje-repl` 的文档字符串。

你也可以通过 Leiningen 的插件 `lein-midje`（像前面提到的在 project.clj 中添加）来运行 Midje。`lein-midje` 允许你以不同的粒度来运行测试，即所有测试、一个组中的所有测试，或一个命名空间中的所有测试：

```
# 运行所有的测试
$ lein midje

# 运行一组命名空间中的测试
$ lein midje com.example.*

# 运行某个特定命名空间中的测试
$ lein midje com.example.t-core
```

## 参阅

- 10.1 节“单元测试”，了解在 Clojure 中进行更多基本单元测试的信息。
- Midje 的 GitHub 代码库（https://github.com/marick/Midje）。

# 10.3 通过随机输入进行彻底测试

作者：Luke VanderHart

## 问题

希望利用随机生成的输入来测试一个函数，确保它在所有可能的场景下都能工作。

## 解决方案

用 `test.generative` 库来指定函数的输入，用随机生成的值来测试它。

要继续本实例，用 `lein-try` 启动 REPL：

```
$ lein try org.clojure/test.generative "0.5.0"
```

假定你打算测试下面的函数，它计算一个序列中所有数字的算术平均值：

```
(defn mean
  "Calculate the mean of the numbers in a sequence"
  [s]
  (/ (reduce + s) (count s)))
```

下面的 `test.generative` 代码为 `mean` 函数定义了一个规格说明（specification）：

```
(require '[clojure.test.generative :as t]
         '[clojure.test.generative.runner :as r]
         '[clojure.data.generators :as gen])

(defn number
  "Return a random number, of a random type"
  []
  (gen/one-of gen/byte
              gen/short
              gen/int
              gen/long
              gen/float
              gen/double))

(defn seq-of-numbers
  "Return a list, seq, or set of numbers"
  []
  (gen/one-of (gen/list number)
              (gen/set number)
              (gen/vec number)))

(t/defspec mean-spec
  mean
  [^example.generative-tests/seq-of-numbers arg]
  (assert (number? %)))
```

要运行 `mean-spec` 规格说明，就调用 `clojure.test.generative.runner` 命名空间中的 `run` 函数，传入运行模拟的线程数，运行的毫秒数，以及指向规格说明的 var：

下面是我们在 REPL 中运行前面例子的情形：

```
(r/run 2 5000 #'example.generative-tests/mean-spec)
;; -> clojure.lang.ExceptionInfo: Generative test failed
```

这显示了生成式测试失败的行为。失败的准确细节被作为 Clojure 信息承载异常（information-bearing exception）的数据返回，你肯定会收到该异常的实例，对它调用 `ex-data` 将返回数据映射表。

在 REPL 中，如果没有明确地捕捉该异常，可以用特殊的 `*e` 符号来取得最近的异常。对它调用 `ex-data`，将返回引起错误的测试用例的信息：

```
(ex-data *e)
;; -> {:exception #<ArithmeticException java.lang.ArithmeticException:
;;     Divide by zero>, :iter 7, :seed -875080314,
;;     :example.generative-tests/mean-spec, :input [#{}]}
```

这表明，仅在 7 次迭代后，使用随机数种子 -875080314，被测函数传入了 `#{}` 作为输入，抛出除数为 0 的错误。

用这种方式突出后，问题很容易发现。如果 `(count s)` 为 0，`mean` 函数将发生除数为 0 的

情况。重写 mean 函数来处理这种情况，修复该缺陷：

```
(defn mean
  [s]
  (if (zero? (count s))
    0
    (/ (reduce + 1.0 s) (count s))))
```

重新运行，显示测试通过：

```
(r/run 2 5000 #'example.generative-tests/mean-spec)
;; -> {:iter 3931, :seed -1495229764, :test testgen-test.core/mean-spec}
;;    {:iter 3909, :seed -1154113663, :test testgen-test.core/mean-spec}
```

这个输出表明，在分配的 5 秒种内，两个线程各自跑了约 3900 次测试迭代，没有遇到任何错误或断言失败。

## 讨论

前面的测试定义中有两个关键部分：defspec 形式本身（定义了生成式测试），以及用于生成随机数据。在这个例子中，数据生成器函数基于 clojure.data.generators 命名空间中的原生数据生成函数。

生成器函数没有参数，返回随机值。不同的函数生成不同类型的数据。clojure.data.generators 命名空间包含了所有 Clojure 原生类型和集合的生成器函数。它也包含了一些函数，随机地从一组选项中选择。例如，前面使用的 one-of 函数，它接受一些生成器函数，从中随机选择一个值。

defspec 宏接受三类形式：一个待测函数，一个参数规格说明，以及一个主体，包含一个或多个断言形式。

待测函数就是要调用的函数。在生成式测试的过程中，它将被调用许多次，每次都用不同的值。

参数规格说明是一个包含参数名称的向量，应该匹配被测函数的签名。每个参数都应该附有元数据。具体来说，它应该有 :tag 元数据键，映射到一个生成器函数的完全限定名称。每次测试驱动器调用该函数时，它将使用每个参数的随机值，该值来自对应的生成器函数。

### 为何用 :tag？

你可能对使用 :tag 元数据有一点困惑。通常，:tag 是一个类型暗示，返回一个 JVM 类。在 test.generative 中，它应该是一个函数，返回你想传递给被测函数的任何类型的值。

> 用这种方式来复用 :tag，主要是出于历史原因。test.generative 主要是受到 QuickCheck 库的启发，该库是用 Haskell 写的。因为 Haskell 是强类型和静态类型的，所以 QuickCheck 确实通过类型签名就足以明白如何生成输入数据。
>
> 在 Clojure 中，这种联系没有那么强，有可能更令人困惑。只要记住，在 test.generative 的上下文中，:tag 不是指实际的系统类型，而是指一个函数，它返回一个类型的对象，而该类型正是你想传递给被测函数的。

defspec 的主体只是包含一些表达式，它们在某些条件不满足时抛出异常。它在每次测试迭代时都被执行，带上可用的、已经实例化的参数，并让被测函数的返回值绑定到 %。简单起见，这个例子只有一个断言，判断结果是一个数字，但你可以有任意数量的断言，执行任何检查。

test.generative 与传统的单元测试之间有一个有趣的区别，它不是指定哪些测试要运行并等待他们运行完，而是指定运行的时间，系统将在这段时间内，运行尽可能多的、随机排列的测试。这样就能保证测试运行时间是确定的，让你能根据情况，在速度和详尽之间折中。例如，你可以在开发时让测试运行 5 秒钟，但每个晚上用一小时的时间，在持续集成服务器上全面锤打该系统，找出可能性为百万分之一的缺陷（毫不夸张）。

### 运行生成式测试

在开发测试时，通过 REPL 运行通常是最方便的。但是，还有许多其他场景（如测试提交钩子（commit hook）或在 CI 服务器上），需要通过命令行运行测试。出于这个目的，test.generative 在 clojure.test.generative.runner 命名空间提供了一个 -main 函数，接受一个或多个目录作为命令行参数，从中找到生成式测试。它在这些位置的所有 Clojure 命名空间中查找生成式测试规格说明，并执行它们。

例如，如果你将生成式测试放在 Leiningen 项目的 tests/generative 目录，就可以在项目的根目录下，运行下面的命令来执行测试：

```
$ lein run -m clojure.test.generative.runner tests/generative
```

如果你希望控制测试运行的强度，可以通过 clojure.test.generative.threads 和 clojure.test.generative.msec JVM 系统属性，调整并发线程数和运行的时间。使用 Leiningen 时，你必须在 project.clj 的 :jvm-opts 键中设置这些选项，像这样：

```
:jvm-opts ["-Dclojure.test.generative.threads=32"
           "-Dclojure.test.generative.msec=10000"]
```

clojure.test.generative.runner/-main 将取得用这种方式提供的参数，并根据它们来运行。

## 参阅

- GitHub 上的 **test.generative** 页面（https://github.com/clojure/test.generative）。
- QuickCheck Haskell 库（http://hackage.haskell.org/package/QuickCheck）。
- 10.4 节“寻找导致失败的值”，探讨了 SimpleCheck，它是一个基于属性的 Clojure 测试库，与 `test.generative` 有一些重叠，还有一些独特的功能。

# 10.4　寻找导致失败的值

作者：Luke VanderHart

## 问题

希望指定函数的一些属性，它们对于所有的输入都保持为真，并找出违反这些属性的输入值。

## 解决方案

使用 `simple-check`（https://github.com/reiddraper/simple-check）。这是针对 Clojure 的一个属性规格说明库，它能够“缩小”输入的情况，找出导致失败的最小输入[1]。

要继续本实例，就在项目依赖关系中添加 `[reiddraper/simple-check "0.5.3"]`，或用 REPL 启动 `lein-try`：

```
$ lein try reiddraper/simple-check
```

然后，找一个函数来测试。这个例子使用了一个编造的函数，它计算一系列数字的倒数之和：

```
(defn reciprocal-sum [s]
  (reduce + (map (partial / 1) s)))
```

下面是测试代码本身：

```
(require '[simple-check.core :as sc]
         '[simple-check.generators :as gen]
         '[simple-check.properties :as prop])

(def seq-of-numbers (gen/one-of [(gen/vector gen/int)
                                 (gen/list gen/int)]))

(def reciprocal-sum-check
  (prop/for-all [s seq-of-numbers]
    (number? (reciprocal-sum s))))
```

注 1：要注意，`simple-check` 找的是“局部”最小，而非“全局”最小，这一点很重要。

seq-of-numbers 是数据生成器，由 simple-check.generators 命名空间中的原生数据生成器构成。

> 不像 test.generative，simple-check 的生成器不是一个返回值的函数，它们更复杂。它们是一些数据结构，不仅定义了随机值如何产生，而且定义了它们如何覆盖“最简单”的可能失败的场景。
>
> 全面讨论创建定制的 simple-check 生成器（而不是原生生成器的简单组合）超出了本实例的范围，但可以在 simple-check 的 GitHub 页面（https://github.com/reiddraper/simple-check）上找到完整的文档。

实际的测试是用 simple-check.properties/for-all 宏定义的，它产生一种属性定义。它接受一种绑定形式（类似于 let 或 for），指定一些可能的值绑定到一个或多个符号，并接受一个主体。主体中实际指定了必须保持的属性，它们必须返回 true，这是一组特定的值能够通过测试的充要条件。

要运行测试，就调用 simple-check.core/quick-check 函数，向它传递定义的属性：

```
(sc/quick-check 100 reciprocal-sum-check)
```

quick-check 接受一些样本和属性定义来执行。属性定义的主体将被反复抽样检查，采用一些随机值，它们绑定到绑定形式指定的那些符号上。

你也许已经注意到，reciprocal-sum 函数有一个问题：如果 0 出现在输入序列中，它会抛出“除数为 0”的错误。quick-check 返回一个数据结构，展示了这个问题：

```
{:result
 #<ArithmeticException java.lang.ArithmeticException: Divide by zero>,
 :failing-size 8,
 :num-tests 9,
 :fail [(5 0 0 -8 1 -2)],
 :shrunk
 {:total-nodes-visited 10,
  :depth 5,
  :result
  #<ArithmeticException java.lang.ArithmeticException: Divide by zero>,
  :smallest [(0)]}}
```

消除 0 值，修复该函数：

```
(defn reciprocal-sum [s]
  (reduce + (map (partial / 1)
                 (filter (complement zero?) s))))
```

重新运行测试，现在它成功了：

```
(sc/quick-check 100 reciprocal-sum-check)
;; -> {:result true, :num-tests 100, :seed 1384622907885}
```

## 讨论

`simple-check` 有一个非常有用的特点，它不仅返回导致测试失败的输入样本，而且返回最小的失败样本。例如，在前面的示例程序中，每次输入序列中出现 0 时，都会导致失败。但是，仅仅查看序列 `(5 0 0 -8 1 -2)`，也许不能明显发现 0 是问题。如果不了解被测函数的细节，也许会觉得问题可能出在负数，或 `5`。`simple-check` 返回的不是任意的失败输入，而是具体的输入，它总是导致程序失败。知道有一个输入会导致失败，这很有用，但更有用的是知道具体导致问题的值。而且，函数的输入越大越复杂，能缩小出错的范围就越有用。

> **test.generative 和 simple-check**
>
> 你可能已经注意到，`test.generative`（在 10.3 节探讨）和 `simple-check` 有许多共同点。它们都生成随机分布的输入数据，它们都指定“成功”的条件，方式都是对所有输入和输出必须成立的属性或品质，而不是具体的实例。
>
> 但是，存在一些关键的区别。`simple-check` 在返回之前让失败的输入最小化，而 `test.generative` 在第一次遇到失败时就失败。但是，`test.generative` 的数据生成器是简单的函数，没有额外指定的行为，这让它们更灵活、更容易扩展。
>
> `test.generative` 不仅能指定测试运行多少次迭代，而且能指定测试运行多长时间，在分配的时间框架中，利用多线程运行尽可能多的测试。
>
> 最后，它们都是有价值的方法，如果你真想彻底测试某个函数，应该认真考虑它们。用哪个应该取决于你自己的具体需求：输入有多大或多复杂，需要对运行时间有怎样的控制，扩展生成原生数据集的可能性有多少。

## 参阅

- 10.1 节“单元测试”。
- 10.3 节“通过随机输入进行彻底测试”。
- `simple-check` 的项目页面（https://github.com/reiddraper/simple-check）。
- “Introduction to QuickCheck”（http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck2），了解启发 `simple-check` 的 Haskell 库的更多信息。

# 10.5　运行基于浏览器的测试

作者：Matthew Maravillas

## 问题

希望运行基于浏览器的测试。

## 解决方案

通过 `clj-webdriver` 库（https://github.com/semperos/clj-webdriver）使用 Selenium WebDriver。这让你能在真实的浏览器环境中，用 `clojure.test` 来测试应用程序的行为。

要继续本实例，就创建一个新的 Leiningen 项目：

```
$ lein new browser-testing
Generating a project called browser-testing based on the 'default' template.
```

修改新项目的 project.clj 文件，像下面这样：

```
(defproject browser-testing "0.1.0-SNAPSHOT"
  :profiles {:dev  {:dependencies [[clj-webdriver "0.6.0"]]}}
  :test-selectors {:default (complement :browser)
                   :browser :browser})
```

接下来，在 test/browser_testing/core_test.clj 中添加一个简单的 Selenium 测试，覆写它的内容：

```
(ns browser-testing.core-test
  (:require [clojure.test :refer :all]
            [clj-webdriver.taxi :as t]))

;; 简单的测试装置，建立一个测试驱动器
(defn selenium-fixture
  [& browsers]
  (fn [test]
    (doseq [browser browsers]
      (println (str "\n[ Testing " browser " ]"))
      (t/set-driver! {:browser browser})
      (test)
      (t/quit))))

(use-fixtures :once (selenium-fixture :firefox))

(deftest ^:browser test-clojure
  (t/to "http://clojure.org")

  (is (= (t/title) "Clojure - home"))
  (is (= (t/current-url) "http://example.com/")))

(deftest ^:browser test-clojure-download
  (t/to "http://clojure.org")
  (t/click {:xpath "//div[@class='menu']/*/a[text()='Download']"})

  (is (= (t/title) "Clojure - downloads"))
```

```
    (is (= (t/current-url) "http://clojure.org/downloads"))
    (is (re-find #"Rich Hickey" (t/text {:id "foot"}))))
```

这个代码库的完整版本在 GitHub（https://github.com/clojure-cookbook/browser-testing）上。在本地签出一份副本，跟上我们的进度：

```
$ git clone https://github.com/clojure-cookbook/browser-testing
$ cd browser-testing
```

在命令行运行测试：

```
$ lein test :browser

lein test browser-testing.core-test

[ Testing :firefox ]

lein test :only browser-testing.core-test/test-clojure

FAIL in (test-clojure) (core_test.clj:20)
expected: (= (t/current-url) "http://example.com/")
  actual: (not (= "http://clojure.org/" "http://example.com/"))

Ran 2 tests containing 5 assertions.
1 failures, 0 errors.
Tests failed.
```

## 讨论

浏览器测试验证应用程序的行为与在目标浏览器中一致。它们测试应用程序的外观和行为，就像在浏览器中渲染一样。

在浏览器中手工测试应用是乏味的重复任务。即使对于普通规模的项目，完成测试所需的时间和工作量也可能无法管理。自动化浏览器测试确保它们一致地运行，并且相对比较快，导致可重现的错误和更频繁的测试。但是，自动化测试缺少人的视觉检查，这是手工测试所固有的。例如，手工测试可以很容易地捕捉一些定位错误，而自动化测试可能就会漏掉，除非明确地对位置进行测试。

要在 Clojure 中编写浏览器测试，就用 `clj-webdriver` 库和你喜欢的测试框架，如 `clojure.test`。`clj-webdriver` 为 Selenium WebDriver 提供了一个简洁的 Clojure 接口，WebDriver 是一个工具，用于控制和自动化浏览器的动作。需要一些附加的配置，才能使用 Selenium WebDriver 或 `clj-webdriver` 和你选择的浏览器。参见 Selenium WebDriver 的文档（http://code.google.com/p/selenium）和 `clj-webdriver` 的维基页面（http://github.com/semperos/clj-webdriver/wiki）。

在深入测试之前，你可以先在 REPL 中尝试 `clj-webdriver`。用 `lein-try` 开始一次尝试

clj-webdriver 的 REPL：

```
$ lein try clj-webdriver "0.6.0"
```

利用 clj-webdriver.taxi/set-driver! 函数，选择 Firefox WebDriver 实现（其他选项包括 :chrome 或 :ie，但它们可能需要更多设置）：

```
(require '[clj-webdriver.taxi :as t])

(t/set-driver! {:browser :firefox})
;; -> #clj_webdriver.driver.Driver{:webdriver ...}
```

这将打开你选择的浏览器，等待接收命令。请尝试 clj-webdriver.taxi 命名空间中的一些函数：

```
(t/to "http://clojure.org/")

(t/current-url)
;; -> "http://clojure.org/"

(t/title)
;; -> "Clojure - home"

(t/click {:xpath "//div[@class='menu']/*/a[text()='Download']"})
(t/current-url)
;; -> "http://clojure.org/downloads"

(t/text {:id "foot"})
;; -> "Copyright 2008-2012 Rich Hickey"
```

在完成后，从 REPL 中关闭浏览器：

```
(t/quit)
```

你的测试将使用这些函数来启动浏览器，并针对它运行。为了省点事，你应该用 clojure.test 的测试装置来设置浏览器启动和结束时的处理。

clojure.test/use-fixtures 让你在每个独立测试前后运行函数，或者将整个命名空间的测试作为一个整体，在它前后运行函数。请使用后一种方式，因为每个测试都重启浏览器会慢很多。

selenium-fixture 函数使用 clj-webdriver 的 set-driver! 函数来启动浏览器，该浏览器由提供给它的关键字指定，然后在浏览器中运行命名空间中的测试，再用 quit 函数关闭浏览器：

```
(defn selenium-fixture
  [& browsers]
  (fn [test]
    (doseq [browser browsers]
```

```
      (t/set-driver! {:browser browser})
      (test)
      (t/quit))))

(use-fixtures :once (selenium-fixture :firefox))
```

使用 :once 测试装置意味着浏览器的状态在各个测试之间是保持的，注意这一点很重要。根据具体应用程序的行为，在编写测试时需要确保这一点，让每个测试从一个共同的浏览器状态开始。例如，你可能删除所有 cookie 或返回某个顶层页面。如果有必要，你可能觉得需要将这种共同的复位行为作为 :each 测试装置。

要开始编写测试，请修改项目的 project.clj 文件，在 :dev 特性描述中包含 clj-webdriver，并在 :test-selectors 中加上方便的 :default 和 browser：

```
(defproject my-project "1.0.0-SNAPSHOT"
  ;; ...
  :profiles  {:dev {:dependencies [[clj-webdriver "0.6.0"]]}}
  :test-selectors {:default (complement :browser)
                   :browser :browser})
```

测试选择器让你单独执行几组测试。这能防止较慢的浏览器测试影响较快的、更经常运行的单元测试和底层集成测试。

在这个例子中，你添加了一个新选择器，并修改了默认值。新的 :browser 选择器只匹配用 :browser 元数据键标注的那些测试。默认的选择器现在将排除带有这种标注的所有测试。

有了测试装置和测试选择器，就可以开始编写测试了。从简单的开始：

```
(deftest ^:browser test-clojure
  (t/to "http://clojure.org/")

  (is (= (t/title) "Clojure - home"))
  (is (= (t/current-url) "http://example.com/")))
```

请注意，^:browser 元数据附在测试上。这个测试被标注为一个浏览器测试，只有在选择那个测试选择器时才会运行。

在这个测试中，就像在 REPL 实验中一样，你导航到一个 URL，检查它的标题和 URL。在命令行执行这个测试，向 lein test 传递附加的测试选择器参数：

```
$ lein test :browser

lein test browser-testing.core-test

[ Testing :firefox ]
```

```
lein test :only browser-testing.core-test/test-clojure

FAIL in (test-clojure) (core_test.clj:20)
expected: (= (t/current-url) "http://example.com/")
  actual: (not (= "http://clojure.org/" "http://example.com/"))

Ran 2 tests containing 5 assertions.
1 failures, 0 errors.
Tests failed.
```

很清楚，这个测试必然失败。用 `http://clojure.org/` 代替 `http://example.com/`，它将通过。

这个测试非常基础。在大多数真实测试中，你加载一个 URL，与页面交互，并验证应用程序的行为符合预期。编写另一个测试与该页面交互：

```
(deftest ^:browser test-clojure-download
  (t/to "http://clojure.org")
  (t/click {:xpath "//div[@class='menu']/*/a[text()='Download']"})

  (is (= (t/title) "Clojure - downloads"))
  (is (= (t/current-url) "http://clojure.org/downloads"))
  (is (re-find #"Rich Hickey" (t/text {:id "foot"}))))
```

在这个测试中，在加载 URL 后，浏览器受命点击用一个 XPath 选择器定位的锚。为了验证预期的页面已被加载，测试比较了标题和 URL，就像在第一个测试中一样。最后，它找到包含版本信息的 `#foot` 元素的文本内容，并验证该文本包含了预期的名称。

`clj-webdriver` 提供了许多其他功能，实现与应用的交互。更多的信息，参见 `clj-webdriver` 的维基页面（http://github.com/semperos/clj-webdriver/wiki）。

## 参阅

- `clj-webdriver` 的 GitHub 代码库（https://github.com/semperos/clj-webdriver）和维基页面。
- Selenium 的项目页面（http://code.google.com/p/selenium）。
- 10.1 节“单元测试”，了解在 Clojure 中进行单元测试的更多信息。

# 10.6 追踪代码执行

作者：Stefan Karlsson

## 问题

希望追踪代码执行，以便了解它在做什么。

## 解决方案

利用 tools.trace 库（https://github.com/clojure/tools.trace）的一群“追踪”函数和宏，在代码运行时进行检查。

在开始之前，先在 :development 特性描述中添加 [org.clojure/tools.trace "0.7.6"] 作为项目依赖关系（在 [:profiles :dev :dependencies] 路径的向量中，而不是 [:dependencies] 路径）；或者，用 lein-try 启动 REPL：

```
$ lein try org.clojure/tools.trace
```

在执行时要检查单个值，就用 clojure.tools.trace/trace 调用来包装该值：

```
(require '[clojure.tools.trace :as t])

(map #(inc (t/trace %))
     (range 3))
;; -> (1 2 3)
;; *out*
;; TRACE: 0
;; TRACE: 1
;; TRACE: 2
```

要检查多个值，又不丢失追踪的上下文，明白追踪的是哪个值，就为 trace 提供一个描述性的名称字符串作为第一个参数：

```
(defn divide
  [n d]
  (/ (t/trace "numerator" n)
     (t/trace "denominator" d)))

(divide 4 6)
;; -> 2/3
;; *out*
;; TRACE numerator: 4
;; TRACE denominator: 6
```

## 讨论

tools.trace 的核心，就是关于反省一段代码的执行。trace 函数是最简单的、最底层的追踪操作。将一个值包装在 trace 调用中将完成两件事：第一是将追踪信息记录到 STDOUT，第二也是最重要的，原封不动地返回原来的值。tools.trace 还提供了其他一些粒度的追踪执行。

从简单值提升一个层次，你可以用 clojure.tools.trace/deftrace 代替 defn 来定义函数，以便追踪该函数的输入和输出：

```
(t/deftrace pow [x n]
  (Math/pow x n))
```

```
(pow 2 3)
;; -> 8.0
;; *out*
;; TRACE t815: (pow 2 3)
;; TRACE t815: => 8.0
```

不建议部署带有追踪的产品代码。追踪最适合开发和调试，尤其是在 REPL 中。请将 `tools.trace` 放在 project.clj 的 `:dev` 特性描述中，让追踪仅在开发任务中可用。

如果你试图诊断一个非常难理解的表达式，请使用 `clojure.tools.trace/trace-forms` 宏来包装表达式，查明异常的原因。如果没有异常，`trace-forms` 就没有输出，正常返回：

```
(t/trace-forms (* (pow 2 3)
                  (divide 1 (- 1 1))))
;; *out*
;; ...
;; ArithmeticException Divide by zero
;;   Form failed: (divide 1 (- 1 1))
;;   Form failed: (* (pow 2 3) (divide 1 (- 1 1)))
;;   clojure.lang.Numbers.divide (Numbers.java:156)
```

除了明确追踪值或函数，`tools.trace` 让你动态追踪 var 或整个命名空间。要为 var 增加追踪功能，就用 `clojure.tools.trace/trace-vars`。要移除这样的追踪，就用 `clojure.tools.trace/untrace-vars`：

```
(defn add [x y] (+ x y))

(t/trace-vars add)
(add 2 2)
;; -> 4
;; *out*
;; TRACE t1309: (user/add 2 2)
;; TRACE t1309: => 4

(t/untrace-vars add)
(add 2 2)
;; -> 4
```

要追踪或取消追踪整个命名空间，就分别用 `clojure.tools.trace/trace-ns` 和 `clojure.tools.trace/untrace-ns`。这将为命名空间中所有的函数和 var 动态地添加或移除追踪。在调用 `trace-ns` 之后定义的所有东西都会被追踪：

```
(def my-inc inc)
(defn my-dec [n] (dec n))

(t/trace-ns 'user)

(my-inc (my-dec 0))
```

```
;; -> 0
;; TRACE t1217: (user/my-dec 0)
;; TRACE t1218: | (user/my-dec 0)
;; TRACE t1218: | => -1
;; TRACE t1217: => -1
;; TRACE t1219: (user/my-inc -1)
;; TRACE t1220: | (user/my-inc -1)
;; TRACE t1220: | => 0
;; TRACE t1219: => 0

(t/untrace-ns 'user)

(my-inc (my-dec 0))
;; -> 0
```

## 参阅

- `tools.trace` 的 GitHub 代码库（https://github.com/clojure/tools.trace），了解追踪函数和宏的完整列表。

# 10.7　用core.typed避免空指针异常

作者：Ambrose Bonnaire-Sergeant

## 问题

希望验证代码正确地处理了 `nil`，消除了潜在的空指针异常。

## 解决方案

用 `core.typed`（https://github.com/clojure/core.typed），它是针对 Clojure 的一个可选类型系统。用它来标注命名空间，检查误用的 `nil`。

要继续这个实例，就创建一个文件 core_typed_samples.clj，并用 `lein-try` 启动 REPL：

```
$ touch core_typed_samples.clj
$ lein try org.clojure/core.typed
```

> 这个实例与其他实例有点不同，因为 `core.typed` 利用磁盘文件来检查命名空间。

例如，假设你要写一个函数 `handle-number` 来处理数据。为了验证 `handle-number` 正确地处理了 `nil`，用 `clojure.core.typed/ann` 标注它接受 `nil` 和 `Number` 类型的联合（`U`），返回一

个 Number：

```
(ns core-typed-samples
  (:require [clojure.core.typed :refer [ann] :as t]))

(ann handle-number [(U nil Number) -> Number])
(defn handle-number [a]
  (+ a 20))
```

在 REPL 中用 clojure.core.typed/check-ns 来验证该函数的正确性：

```
user=> (require '[clojure.core.typed :as t])
user=> (t/check-ns 'core-typed-samples)
# ...
Type Error (core-typed-samples:6:3) Static method clojure.lang.Numbers/add
could not be applied to arguments:

Domains:
        t/AnyInteger t/AnyInteger
        java.lang.Number java.lang.Number

Arguments:
        (U nil java.lang.Number) (Value 20)

Ranges:
        t/AnyInteger
        java.lang.Number

with expected type:
        java.lang.Number

in: (clojure.lang.Numbers/add a 20)
in: (clojure.lang.Numbers/add a 20)

ExceptionInfo Type Checker: Found 1 error clojure.core/ex-info (core.clj:4327)
```

当前的定义是不安全的。check-ns 意识到 + 只能处理数字，而 handle-number 函数接受数字或 nil。

保护对 + 的调用，将它包装在 if 语句中，a 为 nil 时返回 0：

```
(ns core-typed-samples
  (:require [clojure.core.typed :refer [ann] :as t]))

(ann handle-number [(U nil Number) -> Number])
(defn handle-number [a]
  (if a
    (+ a 20)
    0))
```

再用 check-ns 检查该命名空间：

```
user=> (t/check-ns 'core-typed-samples)
# ...
:ok
```

既然现在不会不小心将 nil 传递给 + 了，空指针异常也不可能出现了。

## 讨论

core.typed 的设计目的是避免在类型代码中误用 nil 或 null。要做到这一点，空指针和引用类型的概念是分开来的。这与 Java 不同，在 Java 中，java.lang.Number 这样的类型意味着可以为“空”。

在 core.typed 中，引用类型意味着不可为空。要表达一个可为空的类型（例如前面的例子），就要构造一个*联合类型*，包含期望的类型和 nil。例如，java.lang.Number 在 core.typed 的语法中是不可为空的，联合类型 (U nil java.lang.Number) 表示等同于可空的 java.lang.Number（后面这种表示方法与 Java 类型语法中的 java.lang.Number 最为接近）。

这种概念上的分离让 core.typed 在遇到可能误用 nil 时，抛出一个*类型错误*。前面的解决方案中，在对等价表达式 (+ nil 20) 进行类型检查时，抛出了类型错误。

要更好地理解 core.typed 类型错误，就要注意一些拥有*内联*（inline）定义的函数。core.typed 在类型检查之前，先完全展开所有代码，所以当用户代码调用 clojure.core/+ 时，常常在类型错误中看见对 Java 方法 clojure.lang.Numbers/add 的调用。

在类型错误中还常常看到*有序的交叉函数类型*（ordered intersection function type）。我们的第一个类型错误声称参数 (U Number nil) 和 (Value 20) 不属于任何一个有序的交叉函数类型域，这些域在“Domains”下列出。请注意，提供了两个“Ranges”，它们对应于列出的域。

clojure.lang.Numbers/add 的完整类型是：

```
(Fn [t/AnyInteger t/AnyInteger -> t/AnyInteger]
    [Number Number -> Number])
```

简单来说，该函数是“有序的”，因为它试图匹配参数类型和每一组可能的参数，直到匹配为止。

## 参阅

- GitHub 上 core.typed 的主页（https://github.com/clojure/core.typed）。
- core.typed 的 API 参考（http://clojure.github.io/core.typed/，尤其是核心类型别名的列表，例如，clojure.core.typed/AnyInteger 的条目，http://clojure.github.io/core.typed/#clojure.core.typed/AnyInteger）。

- Types 的维基页面（https://github.com/clojure/core.typed/wiki/Types），记录了合法的类型。
- 10.8 节“用 core.typed 验证 Java 互操作”,以及 10.9 节“用 core.typed 检查高阶函数”，进一步了解使用 core.typed 的例子。

# 10.8 用core.typed验证Java互操作

作者：Ambrose Bonnaire-Sergeant

## 问题

希望验证对 Java 库的使用是安全的、无二义的。

## 解决方案

Java 提供了一个巨大的生态系统，这对于 Clojure 开发者是主要的吸引。但是，从 Clojure 中使用大型的、麻烦的 Java API，常常是复杂的。

要对 Java 互操作调用进行类型检查，就用 core.typed。

要继续本实例，先创建文件 core_typed_samples.clj，并用 lein-try 启动 REPL：

```
$ touch core_typed_samples.clj
$ lein try org.clojure/core.typed
```

> 这个实例与其他实例有点不同，因为 core.typed 利用磁盘文件来检查命名空间。

为了说明，选择一个标准的 Java API 函数，例如 java.io.File 构造方法。

利用圆点构造方法来创建新的文件可能很烦人，所以将它包装在一个 Clojure 函数 new-file 中，接受一个字符串参数：

```
(ns core-typed-samples
  (:require [clojure.core.typed :refer [ann] :as t])
  (:import (java.io File)))

(ann new-file [String -> File])
(defn new-file [s]
  (File. s))
```

在编译这个命名空间时设置 *warn-on-reflection*，这将告诉我们，存在对 java.io.File 构造方法的反射调用。在 REPL 中用 clojure.core.typed/check-ns 检查这个命名空间，将

报告同样的信息，不过是以类型错误的形式：

```
user=> (require '[clojure.core.typed :as t])
user=> (t/check-ns 'core-typed-samples)
# ...
ExceptionInfo Internal Error (core-typed-samples:6)
  Unresolved constructor invocation java.io.File.

Hint: add type hints.

in: (new java.io.File s)   clojure.core/ex-info (core.clj:4327)
```

添加一个类型暗示来调用 `public File(String pathname)`（http://docs.oracle.com/javase/7/docs/api/java/io/File.html#File (java.lang.String)）构造方法：

```
(ns core-typed-samples
  (:require [clojure.core.typed :refer [ann] :as t])
  (:import (java.io File)))

(ann new-file [String -> File])
(defn new-file [^String s]
  (File. s))
```

再次检查，`core.type` 得到满足了：

```
user=> (t/check-ns 'core-typed-samples)
# ...
:ok
```

`File` 还有第二个单参数的构造方法：`public File(URI uri)`。增强 `new-file` 来支持 `URI` 或 `String` 文件名：

```
(ns core-typed-samples
  (:require [clojure.core.typed :refer [ann] :as t])
  (:import (java.io File)
           (java.net URI)))

(ann new-file [(U URI String) -> File])
(defn new-file [s]
  (if (string? s)
    (File. ^String s)
    (File. ^URI s)))
```

即使将输入类型放宽为 `(U URI String)`，`core.typed` 通过 `string?` 谓词，仍然能够推断每个分支拥有正确的类型。

## 讨论

虽然 `java.io.File` 是比较小的 API，但也需要仔细检查 Java 的类型和文档，才能对正确使用外部的 Java 代码有信心。

尽管 File 的构造方法相当无害，但假设要编写 file-parent，它是 getParent 方法的一层薄薄的包装：

```
(ns core-typed-samples
  (:require [clojure.core.typed :refer [ann] :as t])
  (:import (java.io File)))

(ann file-parent [File -> String])
(defn file-parent [^File f]
  (.getParent f))
```

前面的实现避免了反射调用，那么这样就安全了吗？没有。用 core.typed 来检查这个函数是另一种结果。Java 的返回类型是“可为空的”，core.typed 知道这一点。getParent 可能返回 nil，而不是 String：

```
user=> (t/check-ns 'core-typed-samples)
# ...
Type Error (core-typed-samples:7:3) Return type of instance method
java.io.File/getParent is (U java.lang.String nil), expected
java.lang.String.

Hint: Use `non-nil-return` and `nilable-param` to configure where
`nil` is allowed in a Java method call. `method-type` prints the
current type of a method.
in: (.getParent f)

Type Error (core-typed-samples:6) Type mismatch:

Expected:       java.lang.String

Actual:         (U String nil)
in: (.getParent f)

Type Error (core-typed-samples:6:1) Type mismatch:

Expected:       (Fn [java.io.File -> java.lang.String])

Actual:         (Fn [java.io.File -> (U String nil)])
in: (def file-parent (fn* ([f] (.getParent f))))

ExceptionInfo Type Checker: Found 3 errors clojure.core/ex-info ...
```

core.typed 假定所有的方法都返回可为空的类型，所以 parent 标注为 [File -> String] 是一个类型错误。前面每个错误都在说，标注试图将 (U nil String) 声明为 String，最具体的（最有用的）错误排在第一。

core.typed 的设计目的是对 Java 代码持悲观态度，同时又足够准确，避免添加任意的代码来“取悦”类型检查器。例如，core.typed 不相信 Java 方法，所以默认假定所有方法的参数都是不可为空的，并且返回类型是可以为空的。另一方面，core.typed 知道 Java 构造方法从不返回 null。

如果你觉得 core.typed 太悲观，不应该认为返回类型可以为空，那么你可以用 clojure.core.typed/non-nil-return 覆写特定的方法。将下面的代码添加到前面的代码中，类型检查就会成功（简便起见，检查就省略了）：

```
(t/non-nil-return java.io.File/getName :all)
```

在本书编写时，core.typed 没有在运行时强制静态类型重载，所以使用 non-nil-return 和类似功能时要小心。

有时候，类型检查器似乎过分挑剔，在前面的解决方案中，需要两个类型暗示的构造方法。在动态类型的语言中，简单调用 (File. s) 并让反射来解决二义性的问题，这似乎很正常。但是，通过满足 core.typed 的预期，关于构造方法的所有二义性都消除了，插入的类型暗示让 Clojure 编译器能产生高效的字节码。

为什么同时需要类型暗示和 core.typed 标注，才能对有二义的 Java 调用进行类型检查？产生这个疑问很自然。类型暗示是给编译器的指令，而类型标注只是让 core.typed 在类型检查时使用，core.typed 不会影响编译时的解决反射调用，所以它选择假定所有的反射调用都是有二义的，而不会去猜测哪些反射调用可能在运行时解决。这个简单的规格通常导致更快、更明确的代码，在较大的代码集中通常希望如此。

## 参阅

- GitHub 上的 core.typed 主页（https://github.com/clojure/core.typed）。
- core.typed 的 API 参考（http://clojure.github.io/core.typed/），尤其是 non-nil-return 和 nilable-param 的文档。
- 10.7 节“用 core.typed 避免空指针异常”，以及 10.9 节“用 core.typed 检查高阶函数”，了解如何使用 core.typed 的更多例子。

# 10.9 用core.typed检查高阶函数

作者：Ambrose Bonnaire-Sergeant

## 问题

Clojure 强烈建议使用高阶函数，但验证其使用的工具关注于运行时的验证。你希望更早的反馈信息，最好是在编译时。

## 解决方案

用 `core.typed` 对高阶函数进行类型检查。

要继续本实例，先创建文件 core_typed_samples.clj，并用 `lein-try` 启动 REPL：

```
$ touch core_typed_samples.clj
$ lein try org.clojure/core.typed
```

> 这个实例与其他实例有点不同，因为 `core.typed` 利用磁盘文件来检查命名空间。

为了展示 `core.typed` 的能力，先定义一个有类型的高阶函数 `hash-of?`，它接受两个谓词，返回一个新谓词。

用 `clojure.core.typed/fn>` 返回一个匿名函数，并带有类型标注：

```
(ns core-typed-samples
  (:require [clojure.core.typed :refer [ann fn>] :as t]))

(ann hash-of? [[Any -> Any] [Any -> Any] -> [Any -> Any]])
(defn hash-of? [ks? vs?]
  (fn> [m :- Any]
    (when (map? m)
      (and (every? ks? (keys m))
           (every? ks? (vals m))))))
```

`hash-of?` 的每个参数类型都是 `[Any -> Any]`，即一个单参数的函数，接受任意类型，返回任意类型。

验证符合前面类型标注的 `hash-of?` 是正确的：

```
user=> (require '[clojure.core.typed :as t])
user=> (t/check-ns 'core-typed-samples)
# ...
:ok
```

用 `clojure.core.typed/cf` 宏，可以在 REPL（或在测试中）对单个形式进行类型检查。用两个谓词来调用 `hash-of?`，输出得到的类型：

```
user=> (require '[core-typed-samples :refer [hash-of?]])
user=> (t/cf (hash-of? number? number?))
(Fn [Any -> Any])
```

但是，将 + 作为谓词传入，就是类型错误：

```
user=> (t/cf (hash-of? + number?))
Type Error (user:1:7) Type mismatch:

Expected:       (Fn [Any -> Any])

Actual:         (Fn [t/AnyInteger * -> t/AnyInteger]
                    [java.lang.Number * -> java.lang.Number])

ExceptionInfo Type Checker: Found 1 error  clojure.core/ex-info (core.clj:4327)
```

这是因为 `hash-of?` 接受一个带 `Any` 参数的函数，而 `+` 至少要接受一个 `Number`。

## 讨论

要定义能够快速失败的函数，虽然 Clojure 内建的前置 / 后置条件是有用的，但这些检查只能在运行时提供反馈。为什么不对高阶函数也进行类型检查？ `core.typed` 的类型检查能力不只限于数据类型，它也能对函数进行类型检查，就像对类型一样。

用 `clojure.core.typed/fn>` 形式替代 `fn`，来创建并返回一个匿名函数，就有可能利用 `core.typed` 的丰富类型检查系统来标注函数对象。在用 `fn>` 来定义函数时，用 `:-` 操作符来标注参数的类型。例如，`(t/fn> [m :- Map] ...)` 表示一个匿名函数，它接受一个 `Map` 作为其唯一参数。

除了定义，从 REPL 中检查类型也很有用。`clojure.core.typed/cf` 宏是一个多功能的、面向 REPL 的工具，可以按需进行类型检查。它不仅对检查你的代码有用，也可用于研究内建的函数。对任何一个 Clojure 高阶函数调用 `cf`，都会揭示它们的类型签名：

```
user=> (t/cf iterate)
(All [x]
  (Fn [(Fn [x -> x]) x -> (clojure.lang.LazySeq x)]))
```

环绕 `iterate` 类型的 `All` 表明，`x` 中存在多态。念起来是这样的："对于所有类型 `x`，接受一个函数，它接受 `x` 并返回 `x`，并接受一个 `x`，返回 `x` 的惰性序列"。

如果将数目错误的参数传递给一个函数，它又被另一个函数返回，`cf` 宏也可以检测：

```
user=> (t/cf (fn [] ((hash-of? + number?))))
Type Error (user:1:15) Type mismatch:

Expected:       (Fn [Any -> Any])

Actual:         (Fn [t/AnyInteger * -> t/AnyInteger]
                    [java.lang.Number * -> java.lang.Number])
in: ((core-typed-samples/hash-of? clojure.core/+ clojure.core/number?))

Type Error (user:1:14) Wrong number of arguments, expected 1 fixed
parameters, and got 0 for function [Any -> Any] and arguments []
in: ((core-typed-samples/hash-of? clojure.core/+ clojure.core/number?))

ExceptionInfo Type Checker: Found 2 errors  clojure.core/ex-info (core.clj:4327)
```

在这个实验中，对 `hash-of?` 的错误调用包装在一个匿名函数中。在本书编写时，`core.typed` 会先对代码求值，再对它进行类型检查。

没有这一点，原始调用 `((hash-of? + number?))` 就会返回一个普通的 Clojure `ArityException`。

## 参阅

- GitHub 上的 `core.typed` 代码库（https://github.com/clojure/core.typed）。
- `core.typed` 用户指南（https://github.com/clojure/coswre.typed/wiki/User-Guide），尤其是关于多态和函数标注的小节。
- 10.7 节“用 `core.typed` 避免空指针异常”,以及 10.8 节“用 `core.typed` 验证 Java 互操作”，了解如何使用 `core.typed` 的更多例子。

# 关于作者

**Luke VanderHart** 是一名 Clojure 和 ClojureScript 开发者，目前就职于 Cognitect 公司。他是 *Practical Clojure*（Apress）和 *ClojureScript: Up and Running*（O'Reilly）的合著者之一，目前在北卡罗来纳州达勒姆市生活和工作。

**Ryan Neufeld** 通晓多种计算机语言，是一名全能软件开发者，热衷于分布式系统和网络应用开发。Ryan 十分善于为客户解决各种或棘手或简单的软件技术问题，及时为客户交付成果。他目前居住于北卡罗来纳州达勒姆市，是 Cognitect 公司的一名开发人员。

# 关于封面

本本书封面上的动物是土狼（Proteles cristata），一种生活在非洲东部和南部平原的小型哺乳动物，共分为两个种群。虽然其名字在非洲荷兰语中的含义是“土狼”，但它们实际上属于鬣狗科。不像其体型较大的近亲，土狼一般不以腐肉为食，而主要靠长而粘的舌头捕食昆虫（尤其是白蚁）。

土狼全身覆有厚厚的黄色或棕色的毛，间有深色条纹；尾长而毛蓬松，颈部和脊背线上有长长的鬣毛。土狼并不是跑步健将，也不擅长打架斗殴，只能靠身上发达的鬣毛来涨涨威风，吓唬捕食者。土狼的颌骨虽然发达，但其牙齿已退化到只能捕食昆虫，而不能捕食大型兽类。土狼平均体长为 22~31 英寸，平均体重 15~22 磅。

土狼一般夜间出没，白天蛰居于地穴。土狼有极强的领地意识，它们用气味腺分泌物来标记含有自己地穴的领地（一对有交配关系的土狼可能会同时占领多个地穴，轮流使用，但每次只使用其中一两个洞穴）。它们的交配季一般是六月底七月初，孕期 90 天左右，产仔 2~5 只。

土狼有时会被人们误认作鬣狗而杀死，以保护家畜。然而，很多非洲农民却认识到了这种动物的益处——捕食白蚁，控制其数量，从而保护农作物生长。一只土狼一夜便可吃掉 20 万 ~30 万只白蚁。

封面图片来自 Wood 所著 *Animate Creation* 一书。

欢迎加入

# 图灵社区 iTuring.cn

## ——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

**优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。**

## ——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

## ——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

O'REILLY®

# Clojure经典实例

本书涵盖150多个具体实例，展示了有经验的Clojure开发者如何用这门JVM语言完成各种编程任务。解决方案全面广泛：从构建动态网站和应用数据库，到网络通信、云计算、高级测试策略等，面面俱到。这些实例源于全球60多名顶级Clojure开发者。

本书的每个实例不仅可以即学即用，而且其中提供的关于解决方案原理的讨论，让读者可以在模式、方法和技巧上举一反三，从而在遇到本书未提及的其他编程任务时也能游刃有余。

**通过阅读本书，你可以：**

- **掌握内建原生数据和复合数据结构；**
- **使用Leiningen工具创建、开发和发布库；**
- **与本地计算机交互；**
- **管理网络通信协议和库；**
- **掌握连接和使用各种数据库的技术；**
- **应用Ring HTTP服务器库构建并维护动态网站；**
- **解决封装、发布、配置、日志等应用任务；**
- **进行云计算和重量级分布式数据处理；**
- **深入研究单元测试、集成测试、模拟测试和基于属性的测试。**

**Luke VanderHart**是Clojure和ClojureScript开发者，Clojure/core成员，*Practical Clojure*（Apress）合著者之一。

**Ryan Neufeld**是Cognitect公司开发人员，分布式系统和网络应用架构师。

**“Clojure是由实践者所创，也是为实践者而创，而这本书也一样，它源于实践，面向实践，是Clojure实际开发的全面指南。”**

**——Rich Hickey**

Clojure之父，

Cognitect公司CTO

《Clojure经典实例》是全球最优秀的Clojure程序员共同协作的结果，他们的从业背景涵盖航空航天、社交媒体、银行业、机器人、AI研究、电子商务等各行各业。

PROGRAMMING / CLOJURE

封面设计：**Karen Montgomery** 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

| 分类建议 | 计算机/程序设计 |
| --- | --- |

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-39594-8

ISBN 978-7-115-39594-8
定价：95.00元

# 看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks